A Distributed Graph Approach For Retrieving Linked RDF Data Using Supercomputing Systems

by

Michael J. Lewis
B.S. (Purdue University, West Lafayette) 1993
M.S. (University of Illinois at Chicago) 2002

Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2018

Chicago, Illinois

Defense Committee:
Andrew Johnson, Chair and Advisor
Ugo Buy
Ajay Kshemkalyani
Jason Leigh
Venkatram Vishwanath (Argonne National Laboratory)

This is thesis is dedicated to my wife who has endured all my late nights of studying and writing, who has been my top cheerleader! To my mother who has always supported me and to my father, may he be smiling down at me.

ACKNOWLEDGMENTS

I would like to acknowledge the staff at EVL, director Maxine Brown who has always been wonderful to me with advice and encouragement. Dr. Jason Leigh my first mentor and friend, I have learn so much from you. Dr. Andy Johnson who has also been there for me from the beginning and has been so invaluable. I would also like to thank the staff at Argonne National Laboratory and the ALCF department headed by the brilliant Dr. Mike Papka, without their funding for my research and with Maxine helping to make it happen, I would not be where I am today. To Dr. Kshemkalyani and Dr. Thiruvathukal with their guidance and support was so invaluable. Lastly to Dr. Buy and the rest of the committee members who have pushed me to the finish line.

The algorithms represented in this thesis published in the conference proceedings: International Workshop on Semantic Big Data 2017, "A Distributed Graph Approach for Preprocessing Linked RDF Data Using Supercomputers" come solely from my work.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

API	Application Interface
CGE	Cray Graph Engine
DAG	Directed Acyclic Graph
DBMS	Database Management System
GPFS	General Parallel File System
H_2RDF	Hadoop to RDF
LUBM	Lehigh University Benchmark
MPI	Message Passing Interface
NoSQL	Not only SQL
OPS	Object Predicate Subject
OSP	Object Subject Predicate
POS	Predicate Object Subject
PSO	Predicate Subject Object
RDD	Resilient Distributed Datasets
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SOP	Subject Object Predicate

LIST OF ABBREVIATIONS (Continued)

SPARQL	SPARQL Protocol and RDF Language
SPO	Subject Predicate Object
SQL	Structured Query Language
URL	Uniform Resource Locator
XML	Extensible Markup Language
YARN	Yet Another Resource Negotiator

SUMMARY

Many RDF data systems are able to perform queries on different types of connected data structures for a scalable range of input. Partitioning techniques, graph algorithms, and memory based indexing schemes have been heavily researched and integrated into different data systems, in order to produce faster query results with increasing data sizes and different query types. The focus of this work is on two types of powerful (top tier performance in aggregate processing capacity and bandwidth capacity) clustered systems to show conditionally, and definable, time improvements covering dataset preprocessing and query retrieval. Two different algorithmic approaches are used to evaluate query retrieval. One algorithmic approach utilizes a distributed linked data path indexing system to help retrieve queries, the other approach is graph exploration which is finding the linked data at query time according to the connected query patterns. Graph exploration is a common and effective approach used by a number of large scale proprietary RDF systems. In order to implement and evaluate both approaches, the work, called Mantona is developed. Mantona also makes it possible, through generating a preprocessed file *cache-file*, the ability to evaluate performance based on the contents of the *cache-file* and the type of query retrieval algorithm used. This dissertation includes a review of effective RDF query systems and shows the implementation and ramifications of creating a *cache-file* dataset from which the Mantona experiments are conducted over varied processor sizes and query types.

CHAPTER 1

INTRODUCTION

The Semantic Web (1) (2) has become an increasing presence in today's World Wide Web. The web is very useful for retrieving information and accessing pages. Search engines are the web's query agents, keying on vocabulary words to provide a large recall of results but with limited precision. The web is user friendly but not machine processable, meaning a machine does not interpret the meaning of your query statements as a person can extract meaning reading a sentence. A search engine query functions on a word by word basis and provides links to web pages that closely match the searched words without providing the user with contextually precise results. Berners-Lee the original architect of the web (2) envisions the next version of the web to process web searches on a more contextual basis. This new web will extract meaning from the language structure of the query and its association of words in order to provide the user with results that are more accurate and in a more granular form as opposed to an web link to a page that can be full of information that you might not need. This new version of the web is what is called the semantic web. Processing a sentence through its language and grammar can be ambiguous and very complex for a machine to interpret. If a data statement can be reduced to its fundamental associative elements then processing complexity can be reduced. The Resource Description Framework (RDF) is a data model that is based on this fundamental premise of having the lowest denominator of word associations – subject, predicate, object terms or *triples*, in the attempt to extract meaning through the

interconnection of these fundamental associations. From this basic association a subject term has a relation with the object term through the predicate term. The Resource Description Framework Schema (RDFS) provide rules for what constitutes a subject, predicate and object and what defines the interconnectivity among triples in order to provide RDF datasets with contextual characteristics. Many large-scale semantic based query systems came about based on the availability of RDF datasets over the web in order to retrieve more contextual queries. The aim of this work is to show that different types of clustering systems can play a role in RDF dataset preprocessing and RDF query retrievals in ways that have not been researched before. This work conditionally defines these timing improvements based on query, dataset, characteristics over varying processor sizes.

Much of the evolution of RDF systems have focused on and continue to focus on the improvement on retrieval times from datasets that range from an input size of one hundred thousand triples to an input size of over one billion triples respectively. Techniques to transform integer representation of triples to a byte form in order to reduce the number of time-costly data fetches have been used in RDF retrieval systems: RDF-3X (3), and (4). Scan-join systems rely on matching a query clause to a database key or file storage node to access one or more groups of triples (scan) and joining the groups of triple over a common term (join). This dissertation identifies scan-join systems into two types. The first type of scan-join system is file based. Each file has an associated group of triples that have been pre-processed and stored in an associated file. For example, one file can contain all the triples having a common subject, while another file may contain all triples having a common predicate or object term. These

systems look at the most efficient way of joining collections of triples coming from a collection of files that correspond to a query (3), (4), (5), (6). The other type of scan join system uses a key-value, database management systems (DBMS). Each query clause, which represents a pattern of triples, is transformed to a key index. Each key retrieves the triples (represented as the value) within the Database Management System (DBMS) and joins on the collection of key queried triples (7) (8) (9). Data scalable systems (10) (11) (12) use the map-reduce algorithm to query to scale large RDF datasets. Graph partitioning RDF systems (13) (14) utilize graph partitioning algorithms in order to create highly coupled sub-graphs of connected triples for the purpose of reducing node to node communication type. The challenge with these systems is in the defining what associated group of connected triples can be defined as a sub-group that will largely correspond with connected triples stemming from queries. Queries that do not largely conform to any particular subgroup result in a large I/O cost in retrieving parts of the query data for many different computational nodes. RDF graph access systems (15) (16) (17) utilize a distributed memory graph and traverse through connected nodes in order to retrieve query results. Path representation models (18) (19) provide techniques to represent and access connected data structures and triples within an index form. While this approach introduces a way to identify queries through a path based identification, the implementation of this approach is impractical without having a very large system in essence a super-computer to preprocess or cache all the exponential number of paths that are possible given a dataset of triples. Mantona uses a supercomputer to preprocess and store paths up to a manageable depth. Mantona associates a unique preprocessed partial path to processor that will always

match a segment of a query starting from the beginning term, if the partial path does not cover the entire query, graph exploration method used in Neo4J (15), Cray Graph Engine (16) and Trinity (17) will be utilized to cover the rest of the query terms.

1.0.1 Definition of terms

This dissertation uses these set of terms throughout this dissertation in order to explain concepts and definitions regarding query preprocessing, query processing, supercomputer architecture and its components.

1.0.1.1 Supercomputer terms

The following terms are used mostly in chapter two introducing the supercomputer architecture and how it is used in evaluating the experiments in this dissertation.

MPI - The distributed software library used on Mira and Cetus. This library provides the ability to program the sending and receiving of data over processors in different communication topologies in a uniform and non-uniform manner with blocking and non-blocking features.

supercomputer - A top tier computer or cluster with respect to collaborative processing power. This research uses the system at Argonne National Laboratory, Mira. Mira can reach 10-petaflops (10 quadrillion calculations per second) when up to full core capacity.

job - An encapsulation of a program that can be submitted to a cluster for execution. A job includes information such as the maximum length of time the program is to be executed and the number of processors to be used within the job.

qsub - The executable command to submit a job from the supercomputer Cetus, Mira or Cooley.

node - An unit that houses multiple cores. A Supercomputer's total processing ability is defined by the number of nodes it has and the number of cores per node.

core - A processing unit, used interchangeably with processor.

IBM BG/Q - The architecture for Mira/Cetus.

rack - The housing unit, storing the Node cards and I/O Cards and the interconnectivity equipment within the Rack. This is also defined as cabinets. The total computing power of a Supercomputer is equally distributed within the racks.

5D-torus - The topology used to connect nodes together. This is a 2x2x2x2 dimension where each node has 5 neighbor nodes.

rank - A software based representation of a core within a cluster.

blocking - A rank halts until it gets notification that the communication process is complete or a communication error.

uniform/non-uniform - Uniform is when a rank(s) sends or receives the same amount of data to/from the other rank(s) respectively. Non-uniform is when each rank can specify different sizes of data to send to the other ranks. These MPI collective functions have 'v' appended to their name.

onetoone A type of distributed communication where one processor sends data to another processor.

alltoall - A type of distributed communication where each processor sends the same data to all of the processors, and each processor receives data from all of the processors.

allgather - A type of distributed communication where each processor sends data that is uniquely specified to each processor, and receives the complete set of data that has been distributed to all. This type of communication is useful when each processor wants to know what each processor has received, in essence for each processor to see the complete picture, when initially starting with part of the picture.

gather - A type of distributed communication where one specified processor receives a collection of data coming from each processor.

1.0.1.2 RDF, Data Terms

triples - Three tuple entities of data associated with a subject, predicate and object.

triple-root - A triple that serves as the root node for a class of paths that start with this triple at the root.

term - Represents a component within a triple represented as an id or a string.

id - An id is an integer representation of a term id, triple id or a processor id. A processor id refers to the unique processor identification number, also referred to as a rank. This thesis uses the term signature id to specify a hash index, used in map-reduce operations.

vertical partitioning - Partitioning data over processing units based on selected columns of data from tables. In terms of an RDF dataset, a column refers to only subject terms, predicate terms or object terms.

RDF data - Data composed of triples. RDF data can also have a schema (RDFS) associated with the data, specifying the domain and range of terms and conditions for connecting terms together. This dissertation only uses RDF data without a schema.

1.0.2 Processing terms

pattern - A pattern is the schema representation of a group of triples. A pattern is a triple that contains at least one blank node.

path-cache - A tree that contains paths composed of connected triples that can be saved to a file and retrieved.

cache-file A file containing the path-cache of a dataset as well as the term and triple listings, and triple connectivity information.

path-node - A node within the *path-cache*. Each path node contains a list connected triples that match the path characteristics of the node.

triple-product - The list of connected triples within a path-node.

query string - A series of connected patterns of a certain depth in which each pattern must contain one literal and one variable term.

bindings - Bindings are defined as the literal terms that result from the join at the binding variable between two patterns.

Hash joins - A hash join is an iterative join where the resultant of each join is built one pattern to resultant join at a time. Select portion of patterns are loaded into memory in which the bindings are put into a hash map. Upon query time, the resultant bindings are grown with each iteration according to a join plan, in which one pattern is added to the group at a time. On each iteration, two matching binding variables are keyed from the hash map and/or from indexing file storage. The joins are made over the bindings variables and the hash-map at that index is updated.

Merge Joins - A merge join recursively joins subgroups of data, over the overlapping variable attribute starting at the leaf patterns.

1.0.3 Resource Descriptive Framework

RDF (20) is a language/data model used in the Semantic Web community to extract contextual relational and hierarchical data. The core data unit is composed of a three term (subject,predicate,object-value) statement. The representation of subject, predicate, and object is based on the English grammar equivalent. A subject is the main object term that is being categorized or defined. The predicate is the type or action based characterization of the subject, and the object is the noun that the predicate is referring to. Each term represents either a resource or a literal value. A resource is the unique representation of a term. This representation can be commonly linked to a Universal Resource Locator which is a unique web id representation that does not necessarily have to link to a web site. A resource can also be some string that has a specific single context that does not have any web associations. A literal is a string that does not have the requirement of having a single unique meaning or representation. A literal can only be used within the the object term. Subject, predicate and object terms also referred to as *triples*, are able to link to each other like building blocks over its matching terms: subject-subject (s-s) , predicate-predicate (p-p), (object-object) (o-o), subject-object (s-o) object-subject (o-s), predicate-subject (p-s), subject-predicate (s-p), predicate object (p-o), object-predicate (o-p). The most commonly used connections are (s-o), (o-s), (o-o), (p-p) and (s-s) which are used within this work. From these connections, a dataset of triples and terms can be transformed into a type of graph as shown in Figure 1, Figure 2, and Figure 3. This

dissertation expresses URL and literal string terms alike as literals, which in the work qualify as linkable terms. The dissertation also uses terms and entities interchangeably to represent the subject, predicate or object in a triple. Diagram representations of URL terms are abbreviated as *:<resource-name>*, literal strings are in quotations and predicate terms are italicized.

1.0.4 Resource Description Framework Schema

The Resource Description Framework Schema (RDFS) is the meta-data model that gives context and meaning to RDF data. RDFS statements are composed of the same subject, predicate object terms as a triple statement but contain key words within the statement that are used to make further classifications, relationships, and properties (21).

1.0.4.1 RDF and RDFS Key Words

RDF and RDFS key words are used within a triple term to classify a resource, in which RDF compliant query systems can unambiguously apply inferencing rules on and/or make specific categorizations for triples containing the key word. Some common key words are shown below obtained from (21).

1.0.5 Web Ontology Language OWL

The web ontology language adds higher-level meta-data characterizations to an RDF dataset. These include inferencing rules and restriction of classes. The OWL language is representative of an ontology language in that it provides for formal semantics, describing more specifically how data is to be interpreted and categorized. OWL allows for class constructs and class

hierarchies of data, enabling queries for testing for class memberships or if a class construct belongs within a particular class hierarchy. OWL provides the vocabulary to define equivalence of classes and for OWL compliant systems to process consistency detection in order to insure class constructs do not include contradictory logic.

1.0.6 Graph database and graph exploration

A graph database stores its data over a network of linked nodes (containing search-able property,entity information). This is quite different from a relational database that stores its data in multi-column attribute tables. With a relational database, connected data is pieced together through joins over tables. For long conjunctive queries, the application of cross-product operations over each join to link data can be quite costly, especially for large size tables. A graph database search, instead of applying table joins to connect data, follows the paths that match all the clauses of a query. Within a distributed system, a search can be easily processed in parallel. A clustered system can also hash out or partition and pair processing nodes to RDF nodes that are only responsible for path traversals using its assigned starting node. The other advantage of a graph database system is its granularity, meta data and data alike can be simply added as a connecting node to existing node(s) in the graph, without cascading effects as opposed to in a relation database.

1.0.7 Graph Types

RDF based graphs can be represented as an entity graph, schema graph and a triple graph.

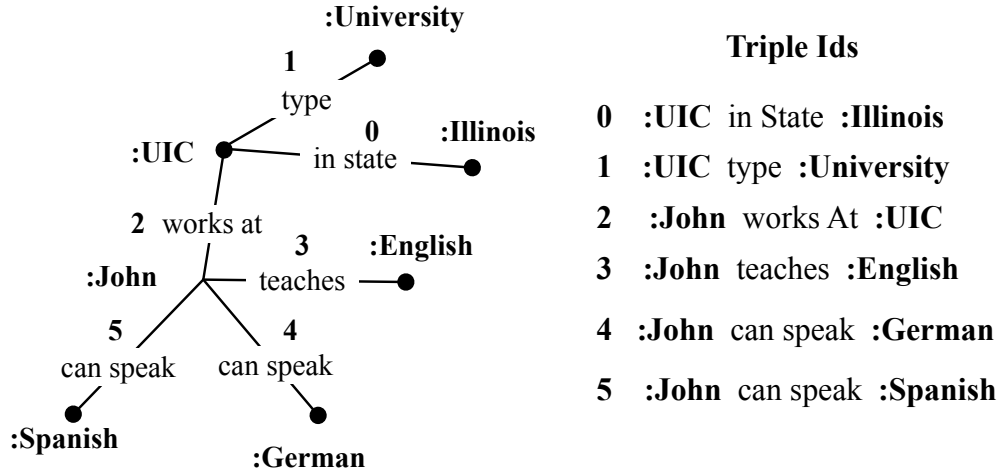


Figure 1: An RDF - entity graph.

1.0.7.1 Entity Graph

The entity graph represents encompasses all of the raw RDF data. In an entity-graph the subject, object entities are the nodes and the predicate terms represent the links. Nodes are mapped to integer ids for fast query processing and reconverted to the string entity representation once the found entity ids are retrieved. An example is shown in Figure 1.

1.0.7.2 Schema Graph

In a schema graph, nodes represent a group of triples based on the types of the subject and object entities Figure 2. This type of graph has some limitations, in that groups, and associations can be unknown, which is the point of querying for the discovery of a group of associated triples. This type of graph layout can be an appropriate mode for RDF systems if types are already understood by the user, or predefined in a schema file.

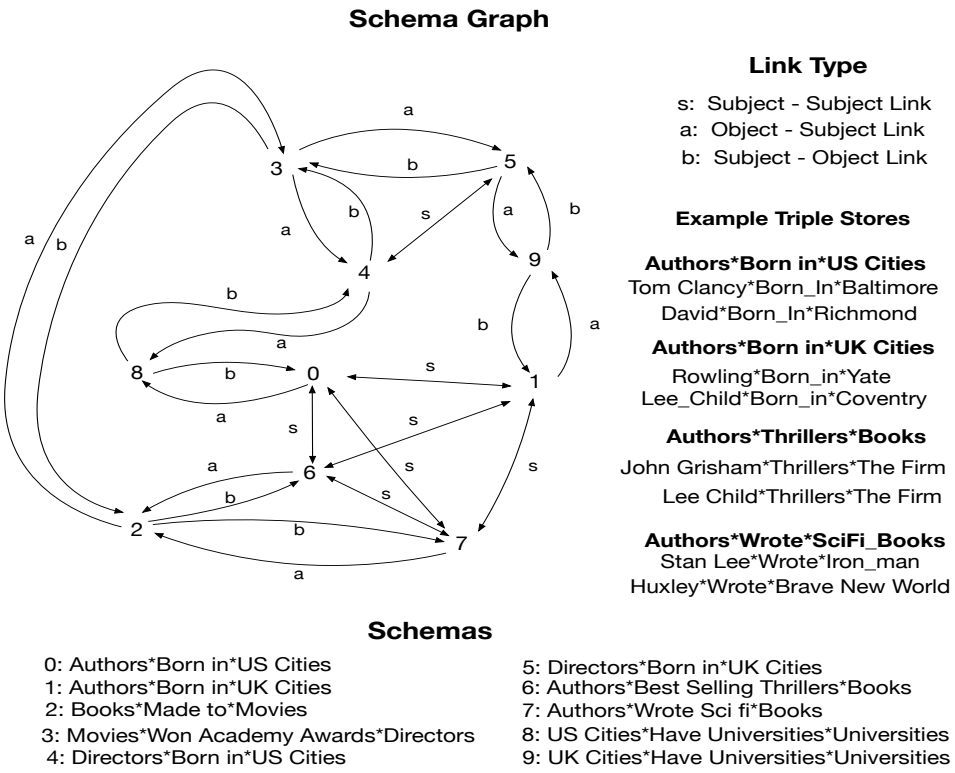


Figure 2: A schema graph and its connections

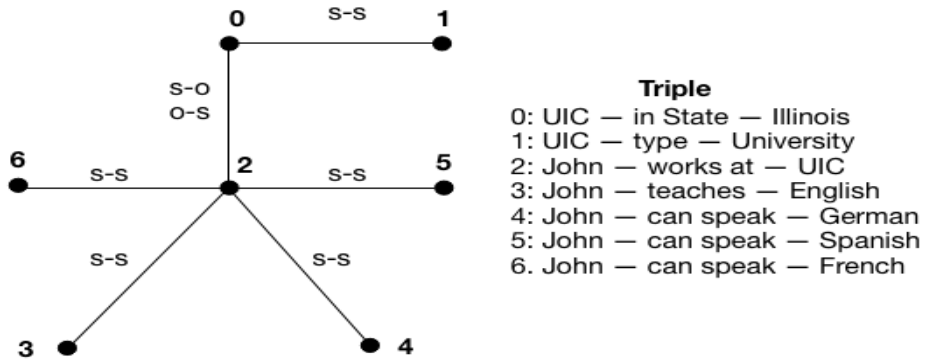


Figure 3: A triple graph.

1.0.7.3 Triple graph

The last type of graph that is used in the work is what this dissertation terms the triple-graph. Triples are the nodes and the connection types (s-s,o-s,s-o,o-o,p-p) between the nodes serve as the links. This work collects and stores the paths of each of the triples graph, given a starting node, or root node. An example is shown in Figure 3.

1.0.8 RDF Queries

A query system takes in a query to implement one or all of the following procedures : (1) retrieving the groups of data based on the query terms in each of the query clauses, (2) filtering the retrieved data using comparison and logical operators (3) applying joins over groups of data (4) selecting which of the resultant data to output, (5) ordering and counting the resultant data, (6) applying database commands such as update, edit or delete on the refined stored data. Query languages SQL (22), SPARQL (23) provide the grammar in which all of these procedures can be implemented on within a query system. This work selectively focuses on how a supercomputer and less powerful large distributed systems can distributively retrieve groups of queried data and implement connected joins.

An RDF query is a collection of query statements. A query statement can represent a URL, value, variable *?var* or blank term *[]*. URL is a unique resource name in which the namespace of the full resource included within the schema or query document and the base of the URL is represented within the query term. A blank term or blank node: *[]* can represent any value as long as there is an existing a triple for it. A query statement that has least one variable or blank term is referred to as a pattern, to signify that multiple triples can belong to it.

For example in Figure 1, a query pattern :John *can-speak* ? (sp?) would include the triples: :John *can-speak* :Spanish and :John *can-speak* :German. Queries containing blank terms are more complex in terms of its low selectivity. With blank terms, query results sizes are larger, however it does not necessarily mean the query extractions come from a broad range of locations within the dataset. Longer queries with low selectivity have another type of complexity where large groups of intermediate data have to be joined. Query input within the work focuses on query selectivity in regards to the number of connected patterns, the number of blank terms and variables within a query. The work does not take in account query complexity in which inferencing and reasoning adhering to a web ontology language is involved (21).

1.0.9 SPARQL

SPARQL (23), an RDF compliant query language offers expressions to satisfy graphical based extractions from linked triples. Dataset timings results from related systems shown in Chapter 7 are a product of SPARQL queries. The two SPARQL queries in Figure 4 are from the dataset of the Billion Triples Challenge.

The first query on the top of Figure 4 shows the latitude and longitude of the Eiffel tower in France. It contains 4 connected query patterns with the merge variable over the subject term. The prefix URL locations are the name-spaces that make each triple term that contains the prefix unique. The first pattern has a blank node on the predicate term, requesting all triples with the object term "Eiffel tower". The second pattern has two literal terms with the prefix geo. The variable ?lat and ?long in the last two patterns are not bounded and work as a blank node. The bounding condition in this query is the common subject over all three patterns.

```
geo: <http://www.geonames.org/>,
pos: <http//www.w3.org/2003/01/geo/wgs84_pos#>,

Select ?lat ?long
Where {
        ?a [] "Eiffel Tower".
        ?a geo:ontology#inCountry geo:countries/#FR.
        ?a pos:lat ?lat.
        ?a pos:long ?long.
}
```

```
geo: <http://www.geonames.org/>,
pos: <http//www.w3.org/2003/01/geo/wgs84_pos#>,

Select distinct ?a ?b ?lat ?long
Where {
        ?a dbpedia:spouse ?b.
        ?a dbpedia:wikilink dbpediares:actor.
    ?b dbpedia:wikilink dbpediares:actor.
    ?a dbpedia:placeOfBirth ?c.
        ?b dbpedia:placeOfBirth ?c.
        ?c owl:sameAs ?c2.
    ?c2 pos:lat ?lat.
    ?c2 pos:long ?long.
}
```

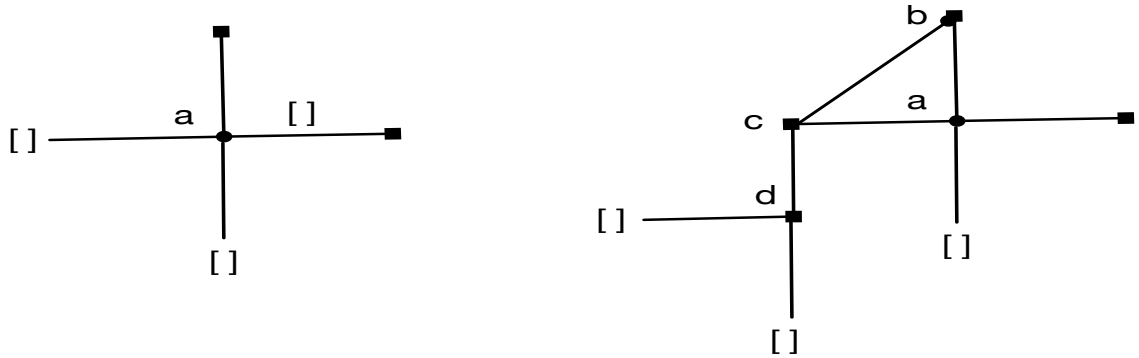


Figure 4: Two SPARQL queries and the corresponding query graphs.

This is a query based on only one variable that serves as the merge point. The second query in Figure 4 requests the latitude and longitude and the names of two married actors with the same place of birth. There are four different variables that serve as the binding connections between patterns with subject-subject and object-subject/subject-object bindings.

1.0.10 Query graphs

There are multiple factors that show the complexity of a query. A visualization showing the query connections and the pattern types can be used to quickly understand the linking factors within a query. This dissertation uses query graphs, shown at the bottom in Figure 4 , to show query complexity for each query experiment conducted in Mantona and the related experiments reported in the results chapter.

1.0.11 The Mantona query system

This dissertation introduces the work called Mantona. Mantona is an RDF query processing system, written in C++ using the Message Passing Interface (MPI). Mantona is able to preprocess conjunctive triple-triple connections, and utilize these store joins to expedite query retrievals. The contribution of the work is the following.

1. The ability to effectively use a supercomputer to generate and store triple ids, term ids and connected data within a single file. Query systems need to process the original dataset to map the string terms to integer ids and produce connectivity information to possibly cache. For standalone, low processor count query systems taking in large datasets containing billions of triples may take hours to process. This research shows a scalable improvement in processing time through the increasing of the number of processors. This

research shows also that any type of clustering system can partition the cache-file over its number of processes for faster loading.

2. A graph-retrieval system, designed to retrieve conjunctive pattern based queries by indexing the join data from the graph structure that match query patterns. This research shows from side to side experiments within the Mantona framework (graph retrieval vs. graph traversal) over different datasets that implementing a distributed graph-retrieval systems reduces query timings over certain type of queries as opposed to graph traversal methods.
3. With Mantona, one can analyze the different types of queries to determine how ordering, pattern selectivity and query connectivity have an effect on query timings within multiprocessor systems.

The dissertation is broken into 9 chapters with Chapter one being the introduction. Chapter two discusses retrieval techniques of related RDF query systems. Chapter three shows how a supercomputer environment and its distributed software can be utilized for query processing and preprocessing. Chapter four details the process of mapping strings to term ids, generating a unique list of term ids and triple ids, generating an adjacency list of connecting triples stemming from a root triple, and loading raw datasets and cached files. Chapter five reviews *cache-file* the creation and the implementation of the Path-Cache algorithm and its complexity. Chapter six details the Mantona query retrieval algorithms: *path-cache* and *graph-exploration*, and the implementation of the query generation algorithms. Chapter seven reviews the results from related systems and through Mantona over dataset size, processor count, and query configura-

tion. Chapter eight shows the experimental results using the LUBM dataset on Mantona, and compares the Mantona results with the LUBM results from related systems. Lastly, chapter nine provides the conclusion.

CHAPTER 2

RELATED SYSTEMS

This chapter reviews the works of RDF query systems with reference to how queries are collected. This chapter reviews scan based retrievals using different configurations of table and database structures, graph exploration methods, query retrieval using map reduce, research that utilizes a path index to represent conjunctive queries, and optimization techniques that speed up the join, query retrieval process.

Section one provides a review of scan join systems: Hexastore (6), RDF-3X (24), Acumulo (25), AMADA (8), Casandra, and C-Store (26), (27). A scan-join process is used for single processor systems, or implemented within a distributed system in which each processor implements the scan-join query retrieval process for its section of data.

Section two covers map-reduce systems: Shard (10) HadoopRDF (28), H2RDF (29), RAPID (30), and (31). The map-reduce algorithm is used to iteratively process joins in coordination with a join plan of the query graph. The different type of map-reduce based joins are also discussed within this section.

Section three covers graph-exploration query retrieval process. In a graph-exploration query retrieval process, each processor has a data node start point, and traverses across links to neighboring nodes that are in match with the query pattern. If the summation of explored paths match the query graph then that query result is accumulated. This section provides reviews of systems: Trinity (17), Neo4J (15) and Cray Graph Engine (16).

Section four covers path indexing research produced from Yamamotoa et al. (32), Matono et al. (19), Groppe et al. (18) and the work from GRIN regarding the GRIN index representation (33). A path indexing only solution is not feasible to implement within large datasets. Mantona implements a partial indexing as a part of its query retrieval algorithm.

Section five covers optimization techniques through the adoption of compression, query planning and join techniques that can expedite query retrieval.

A table is shown to chart the features of the different RDF query systems and query results are listed in a table to provide a context of query retrieval times based on query features of link and length. Lastly, a chapter summary is provided to give the final word of the role each query system provides in the realm of RDF query frameworks and to show where the Mantona research fits within this space of query frameworks.

2.1 Scan Join Systems

Scan-join is a two-step process indexing the literal query pattern terms over a database or file system then applying a join on the overlapping triple patterns. Each scan-join system has its own particular niche on how to look up and store its triple patterns. Triples can be grouped and sorted over a particular term or grouped over a particular literal term (property table). The common specification of a triple group and its ordering is the listing of the terms by the first letter of the term. The table: POS would list all the triples sorted by the predicate term then object then by subject. By having multiple tables such as POS SOP OPS a system can scan the table that can be filtered to match the pattern the fastest. the last step is the joining of the queries data.

2.1.0.1 Hexastore, BitMat

Hexastore (6) indexes a query pattern based on subject, predicate or object. From the index term, Hexastore accesses one of six based tables pos, spo, osp and ops in order to materialize the query pattern. BitMat (4) also provides pos, spo, osp and ops tables, however BitMat tables are generated in the form of a bit cube where each axis of the cube represents a term type.

2.1.0.2 Property table, vertical partitioning

Besides providing different column sorted combinations of storing triple data, systems: Jena (34), Sesame (35) C-Store (26), (27) and MonetDB over XML data (36) offer both property tables and vertical partitioning as a solution to access the queried data. For queries centered around a term ("bush queries"), these systems would search for property tables. The system would have tables based on the most commonly found subject, predicate or object data and sort the table based on the highest frequency term. The vertical partitioning approach would produces tables that would link parts of a multiple attributed values to a particular subject. For example if a dataset structure was more subject themed with schema [Artist,Book,Release Date,Date Of Birth] tables can be stored by [Artist, Book], [Artist, Release Date] and [Artist, Birth Date] if queries were found to be more associated with a subject and a particular property of that subject.

2.1.1 Key based scan

Another type of scan-join systems uses a key for as the scan phase. A key represents one or two terms RDF-3X (24) system or a condensed representation of a two-dimensional term space.

RDF-3X is a query-processing engine that utilizes a RISC architecture in order to facilitate a faster processing of merge joins (24). RDF-3X includes a query optimizer that chooses the optimal join order. RDF-3X (24) uses a B+-tree with a single index scan all 6 different permutations of triple patterns in : (S)PO, (S)OP, (O)SP, (O)PS, (P)SO, (P)OS. The term in parentheses designates the key that is scanned. The resulting triples found from a scan are put in lexicographical order based on the order of the resulting terms; (S)PO with S representing the key. The resulting triples are ordered by the common subject followed by the predicate term then object term.

Accumulo (25) is an open source key-value database system. Its design is based on Google's BigTable (37). Accumulo provides random, real time, read/write access to large datasets in a distributed environment using commodity hardware. Accumulo provides sorting of keys in lexicographical ascending order. Each key packet is encapsulated with meta data, which includes key id, row id, column identifiers, time-stamp and value. Accumulo also provides a level security mechanism for clients. In addition, Accumulo provides a *Batch Scanner* client API that condenses all range-scan requests into one scan. This feature is particularly helpful for RDF access involving access and linking to different triple stores required in a conjunctive query.

Casandra is a key-value data store that is also modeled after BigTable (37). Both systems use a nested indexing key structure *key : value* ; the value can be another key : value pair as in *row key : superColumn key :value* or an actual value or empty value, denoted as -. In Casandra, the super column indices can be sorted or mapped to a hash value.

AMADA (8) uses Amazon's simple storage service (38) to process a range of triple patterns covering SPO,OSP,POS. H2RDF (29) is a distributed triple store that combines the Map-Reduce framework with the NOSQL key, value store of BigTable. H2RDF materializes on 3 of the 6 different spo permutations (*spo*,*pos* and *osp*). CumulusRDF (7), implemented on top of Cassandra (39) uses nested s,p,o indices. Cassandra using the nested index can cover a range of triple patterns: *spo*, *sp?*, *?po*, *s?o*, *?p?*, *s??*, *??o*, *???*.

2.1.1.1 Graph Partitioning Systems

The cluster based RDF systems GRIN : (33), WARP(14) and Partout (13) use a scan based algorithms within a cluster system. Cluster based RDF systems use partitioning algorithms to store regions of triples based on its community of neighbor nodes. Partout uses the partitioning tool METIS (40) to find the k number of partitions that created the least amount of edges among each other. Linked queries however can be unpredictable to predict, unlikely triples can be found to be connected to each other through a series of s-o,o-s connections. An overlapping strategy of duplicating triple nodes over processors is used in (10). These nodes represent a shared link of connections across processors, but this technique can only offer a short-range solution for pattern-linked queries.

2.2 Map-reduce

The common use for map reduce with regards to processing RDF data is in the full utilization of all processors for joining query data. Map-reduce has also been used for join evaluations as proposed in (41), (42),(43), (44). The two types of joins that are most widely used are standard reparation join and the broadcast join. The standard reparation join distributes the two triple

products on the join key during the shuffle phase and joins them in the reduce phase. The broadcast join is a map node only procedure. Each node broadcasts their materialized patterns to all nodes as each map node grows its triple product.

Shard (10) was the first system to use an iterative-join query plan with Map-reduce. The Map-reduce job is iterated n times in which n is the number of query clauses consisting of the conjunctive triples patterns. The map nodes assign and key variable bindings to triples, partitioned from an RDF file and the reduce node joins the triple patterns over the common variable binding, and removes duplicates. The last iteration removes redundancies and applies a projection on the resultant data, Figure 5.

Pig Latin (45) provides users with a language similar to nested relation algebra containing primitives such as join, filter and union. This language is then compiled into MapReduce to correspond with a iterative-join query plan. In PigSPARQL (46) a SPARQL query is translated into a iterative-join query plan. The frameworks HadoopRDF (28), H2RDF (29), and RAPID (30), (31) evaluate merges of triple stores under MapReduce with the goal of reducing the number of iterations of a MapReduce job. HadoopRDF and H2RDF uses a heuristic to add as many joins to a job thus providing fewer MapReduce iteration. RAPID adopts a merge join approach. RAPID translates join tree structures into grouping operations thus creating a shorter path and creating more bushy sub-queries. RAPID provides a data model and algebra called Nested Triple Group Algebra (47). Each of the intermediate data within a iterative MapReduce job are factored within a triple group. In (47) the Nested Triple Group technique is used to reduce scans when there are repeated properties within the query graph. In (48) Map

nodes are only used in the scanning and joining of triples. Triples that share a common variable are grouped and processed in a single Map iteration. Scalable Sparql (12) uses MapReduce for query coordination and processing and RDF-3X for data accessing over a federated distribution of processors.

2.3 Graph exploration systems

Graph exploration systems require an efficient node-to-node communication and distributed system that can hold a large amount of memory. A graph exploration system traverses the path of a query request over its distributed network of nodes. Each node holds a section of terms and its related term connections as well as a link to the processor of its neighboring term. Because of this linked traversal behavior of graph exploration systems, the underlying distributed framework must be designed to have some type of fast, possibly memory-to-memory connections across processors in order to minimize data accessing time over processors. RDF data must be cached in memory in graph-exploration systems. A long conjunctive query can visit many different processors depending on wherever the next node link points to. If for each node visited there was an I/O request for data, at the very least there would be a I/O reduction based on the long time it takes to access storage as opposed to memory and at the very worst it could deadlock or dramatically slow the system with different nodes blocking to request data from the same storage segment. This section reviews the graph exploration software systems (Cray Graph Engine, Spark, and Trinity) that account for these I/O processor to processor communication problems and make it easier for a developer to implement graph based query retrieval algorithms. It has been shown through query timings and data sizes that graph

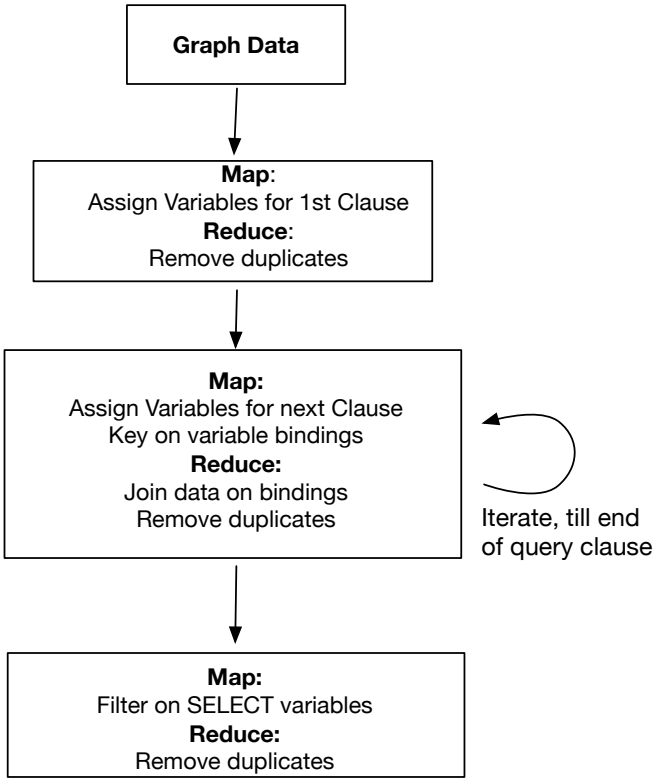


Figure 5: Using Map Reduce to join on bindings.

exploration algorithms are processor scalable and have been able to hold up to 1GB in triples for robust distributed architectures.

2.3.0.1 Cray Graph Engine

The Cray graph engine (16) uses an RDF graph database over a distributed scalable processing environment and uses graph algorithms, pattern analysis and filtering methods as a means to infer on data relationships from SPARQL queries. The Cray Graph engine is supported by scheduling and cluster management software. YARN (49) used for HADOOP , Mesos is the cluster management software that also houses the Cray Graph Engine and Marathon (50) is used as the container orchestration platform. The challenge for acceptable retrieval times for graph-based searches comes from the processing time assembling the data through the process of path traversals in which entity points may not reside on the same processor. CGE uses a shared memory architecture: Urika-GD (51) in order to achieve fast access times when indexing linked data. The Cray graph engine is compatible to SPARQL queries and adds some additional features. CGE allows for a variable representation of a an expressed sub-graph and can apply a set of graphing algorithms e.g. community detection to that subgraph, all encompassed within the SPARQL query.

2.3.0.2 Spark, Graph-X

Graph-X (52) is the graph engine for Spark. Spark is an open-source data-parallel, fault tolerant computation platform that can facilitate an in-memory Map-Reduce paradigm. It uses an in memory architecture for fast transfers of data across processors within the cluster. Spark (53) allows the developer to instantiate fault tolerant memory components – Resilient Distributed

Datasets (RDDs) that functions as application data containers. RDDs come with an API where the developer can transparently apply different grouping operations to the represented data e.g. (map,group-by,hash-join). Cascading operation can also be applied to an RDD. For example a developer can create a cascading map-reduce command within one line. Spark can operate interactively through a Scala scripts and is also Java and Python compatible. A unique feature of Spark is its ability to automatically recreate RDDs upon failure through lineage. Lineage is a type of providence for RDDs; each RDD has a graph representation of meta-data specifying how the RDD was created. If an RDD fails, it can automatically be reconstructed through its lineage. Graph-X is the graph module for Spark. It contains its own graph specification language where one can specify the adjacency list structure of the graph as well as the vertex data. Graph-X adopts a vertex cut as the way to partition each vertex and edge across the cluster. A vertex is shared over each partition that is represented at the cut; edge information is bound to a single partition within the cut. Graph-X assigns its partitions by assigning its edges to partitions in a way that minimizes the number of vertices that are shared across processors.

2.3.0.3 Trinity

Trinity has shown through large-scale experimentation using LUBM (54) and DBpedia (55) generated datasets to outperform RDF-3X (3) and BitMat (4). Trinity stores its data in a memory graph where nodes are the individual triple terms. Each node has an adjacency list of incoming and outgoing neighbor nodes. The collection of graph nodes residing on an individual processor are grouped together based on a SPO or OPS index.

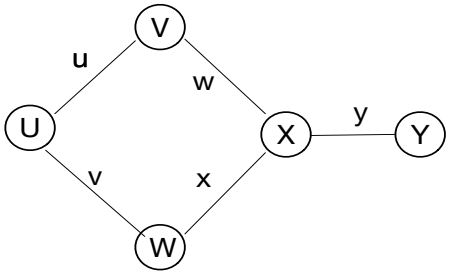
2.4 Path Based Indices

The connection between path indices and preprocessed joins is in the creation of an organizational structure to index paths of connected data. Early research initially covered by Yamamotoa et al. (32) created structures for generating path indices from XML documents. Matono et al. (19) proposed a technique for translating RDF path expressions into suffix arrays using Directed Acyclic graphs (DAGS) extracted from an RDF dataset and/or schema. All paths and links are individually labeled with a character, and put into a numerical grid, in which every path has a unique two-dimensional index. Paths are then put into lexicographical order, and duplicates are removed in Figure 6. In Groppe et al. (18) joins were indexed in a hash-map over s-s,s-p,s-o,p-p,p-o and o-o connections for one join, two triple patterns and multiple joins over multiple triple patterns. With Grin (33) the RDF-graph is partitioned over center nodes that adhere to a particular index. Using a center indexing formula queries can be determined if component lies within the radius of any center node.

2.5 Optimization Techniques

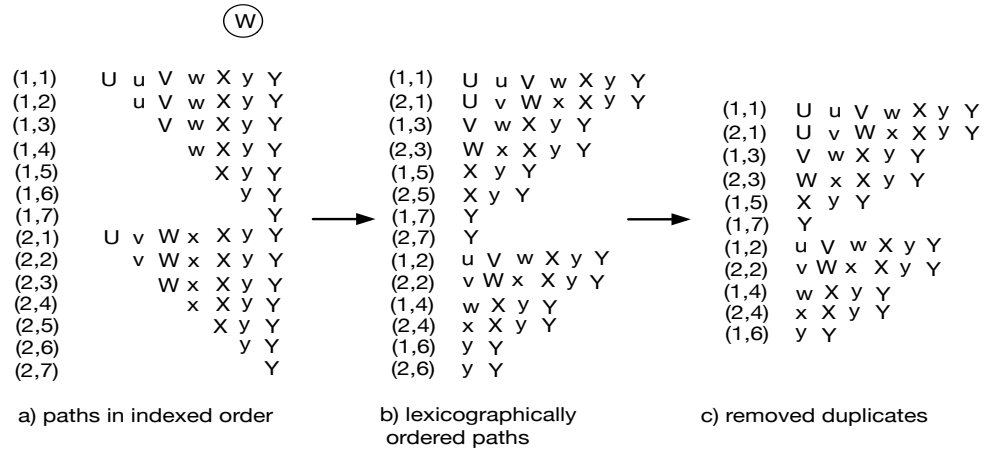
2.5.0.1 Compressed triples

Compressing triples can be useful in retrieving more triples faster. Each triple stored in the B+-tree is compressed into a maximum 13 byte package. The actual triple is not stored but instead the offsets of each term are stored with respect an triple-index within the group. The first byte, the header, specifies the number of bytes to record the offset size of the subject, object and predicate. The efficiency of the compression is based on the assumption that within each



	1	2	3	4	5	6	7
1	U	u	V	w	X	y	Y
2	U	v	W	x	X	y	Y

(a)



(b)

Figure 6: (a) Indexing paths to a grid index. (b) Ordering, sorting and removing path duplicates.

The table below shows the common query retrieval and storage characteristics from a list of RDF query systems.

$RDF - Systems$	Vertical Partition	Map Reduce	Graph Partition	Graph Access	Path Index
HexaStore	X	-	-	-	-
Jena	X	-	-	-	-
Sesame	X	-	-	-	-
C-Store	X	-	-	-	-
Amada	X	-	-	-	-
H2RDF	X	-	-	-	-
CumulusRDF	X	-	-	-	-
Trinity	-	-	-	X	-
Grin	-	-	X	-	X
RDF-3X	X	-	-	-	-
Matono et al	-	-	-	-	X
Scalable Sparql	X	X	-	-	-
Shard	-	X	-	-	-
HadoopRDF	-	X	-	-	-
H2RDF	-	X	-	-	-
RAPID	-	X	-	-	-
Partout	-	-	X	-	-
BitMat	X	-	-	-	-
Neo4J	-	-	-	X	-
CGE	-	-	-	X	-

TABLE I: Features of RDF systems.

$RDFHelper - Systems$	Key Storage	WorkFlow Systems	Partition Systems	Comm. Based
Spark	-	X	-	-
MPI	-	-	-	X
Hadoop	-	X	-	-
BigTable	X	-	-	-
Apache Accumulo	X	-	-	-
Dynamo DB	X	-	-	-
HBase	X	-	-	-
AWS	X	X	-	-
METIS	-	-	X	-

TABLE II: Features of frameworks used by RDF systems.

triple pattern, all triples values slightly deviate from each other with respect to lexicographical proximity.

2.5.0.2 Join optimization

The selection of join order within a query system, can make a significant difference in query time retrievals. Join order is the order of join connection from the query graph. The same join type over the same connected triple patterns will output the same result for different join orders. The join order that reduces the number of join matches early, reflect the join order that uses the less amount of computations. RDF-3X can get query results for a single connected query graph, or a disconnected query graph. From a disconnected graph, based on the type of query the results of the two graphs are combined. The technique used to find the optimum join plan is through bottom-up dynamic programming using a cost estimate based on selectivity estimates. A selectivity estimate determines through statistics, the probability of a pattern or

individual term coming up in a query. RDF-3X proposes two types of statistics; one is to create a histogram of pattern types. Each bucket in the histogram stores a pattern containing a one or two term literal. The bucket information includes the cardinality of the triple pattern, the number join types that can be connected to the pattern. The second type of statistics is from storing the most frequent paths. Each path is composed of sub-paths and each sub-path is its own path containing its own selectivity number from the cardinality of the connected joins. The problem with the first method is that the cardinality statistics over a pattern does not take in account connectivity information in how like would a pattern connect to other patterns. The problem with the second selectivity method is that not all path combinations are covered, and for those that are not covered, the histogram method independent of connections is used.

2.5.0.3 Join techniques

The join algorithm is the following: create a map entry for all the variables in the query. The key represents the variable for a clause, the map value is the bindings associated with that key. Pick a triple pattern stemming from a query clause and put the associated bindings within the variable map. Traverse through every other query clause that contains the same variable and produce its bindings. If there are no bindings for that variable that match the current binding entries within the map, then remove that variable key as a map entry. If there is a triple pattern that refers to multiple variables and one of the variables was removed from the map, then the other variable must be removed from the map. Continue to remove variables based on this cascading procedure. Find the next triple pattern and conduct the same procedure on a

variable. After all the triples patterns have been examined and bindings or variables extracted, then the variable that are left and the associated binding are the return results of the query.

2.5.1 Query planning

Query planning involves picking join combinations over triples patterns for vertically partitioned systems or node exploration for graph retrieval systems that result in choosing a path that has the minimum number of computations. Picking the "right" triples or nodes in a path depends on some statistical determination over frequency of connections or using the inherent data correlations that can be determined. Stocker et al (56) holds subject, predicate and object terms as independent entities. The selectivity (connections) of each term is gathered and the triple selectivity is determined by the product of the triple term selectivities. In (17) a graph exploration is done to find the cost of each SPO or OPS group of triples at each node. The formula is the following:

$$|R(q)| = |B(src)| \times C_p/C_p(src)\,|B(tgt)| = |B(tgt)|C_q(tgt)/C_q(src)$$

The variables , C_q(subject), and C_q(object) denote the binding size over the connecting nodes. The C_p denotes the binding size for that particular graph node on the supercomputer. B(tgt) is the binding size of the target nodes, R(q) is the estimated size of the results. Query cost then can be determined as a linear combination between R_q and B_src.The shortest cost path can then be determined by dynamic programming.

$$C' = minC', C + costq$$

The C' represents the current cost and the q represents the cost for connecting to a target node, this node can be isolation or a part of another tree.

In Husain, Mohammad, et al (57), map-reduce jobs are used to develop a greedy based query plan. A query graph is produced in which the source node represents the list of variables in the connected join produce of triples. The target node denotes the resultant list of variables produced after a map reduce job. The cost is the following.

$$\sum_{i=1}^{n-1} Job_i + MI_n + MO_n + RI_n$$

$$Job_i = MI_i + MO_i + RI_i + RO_i$$

In which MI_i = the map input stage for Job i, MO_i = the Map output phase for Job i. RI_i = Reduce input stage for Job i. RO_i Reduce output phase for Job i. Cost is based on the number of triples used in each phase.

RDF-3X uses map-reduce as a partitioning strategy. Each iteration places overlapping RDF graph nodes to a partition. Each iteration adds an extra hop to the partition.

2.6 Summary

The different amount of query solutions ultimately stem from the type of dataset and the type of queries that are presented. For small datasets and datasets that were predispositioned for SQL based queries, a scan-based solution approach is useful. The more powerful scan based RDF systems would take advantage of utilizing large-scale database systems such as BigTable and Amazon's simple storage service as a tool for scanning the data for triples. However scan

based solutions do not provide a method for joining the groups of triples received. Large datasets that could produce many link queries would require some type of join procedure and processor coordination to reduce the join computation time. Map-reduce systems provided a data-scalable solution for these tree conjunctive queries and can have different joins processing on different processors based on a the join hash key. Graph partitioning systems provided solutions for transforming and partitioning the dataset into a collection of sub graphs of connected triples. However, this could not give a consistent solution for graph queries that could touch many different sub-graphs. A Path indexing representation provided a uniform way to uniquely specify a conjunctive query result, the problem was this could not be feasible done base on the exponential amount of linked graph data that could be produced from datasets as low as 500,0000 triples.

CHAPTER 3

SUPERCOMPUTER ENVIRONMENT

3.1 Introduction

Mantona was developed under the resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. The Supercomputers used for this research are named Mira and Cooley. This chapter provides a brief overview for the supercomputer architecture for Cetus/Mira and list of resources for Cooley. This chapter also reviews the communication features of the MPI library that is used in Mantona, and provides an overview of the job-queue. A summary is provided at the end of the chapter.

3.2 Mira

The Mira supercomputer operates on the Blue Gene Supercomputer architecture BG/Q. The BG/Q system contains 48 racks, each rack contains two mid-planes containing 16 node cards a piece and an I/O drawer containing 8 I/O node cards, Figure 7. Each I/O node card is responsible managing the transport of data to and from the storage systems. Every two-node cards has access to one I/O node card through at 2GB/s optical link. Node card to node card communication also uses an optical link. Each node card contains a 2x2x2x2x2 electrical interconnect to connect to each of the 32 compute cards (node) and uses a a 5D torus topology

over the node-node network. A node contains 16 cores using a PowerPC A2 16000MHz processor and contains 16 GB of DDR3 ram.

Mira allows users access to a General Parallel File System (GPFS) with 24 PB of capacity and 240 GB/second bandwidth. Mira has connection to the Energy Science Network or ESnet. This network provides high performance connections using 100 gigabit per second links (58).

3.3 Cetus

Cetus links to the same computing resources and Mira, but is only allowed to use up to 4,096 nodes having the limit of 16 cores per node. Cetus also can only schedule jobs that are limited to one hour.

3.4 Cooley

The Cooley supercomputer has a much higher memory capacity per node than Mira, but has few computing nodes. Cooley has a total of 126 nodes, with each node having 12 cores and one NVIDIA Tesla K80 dual-GPU card. Each node contains 384 GB of memory and 24 GB of GPU RAM. Access to Cooley is provided by two login nodes, which provide compilation and job submission capabilities (59).

3.5 Communication Library

The Message Passage Library (MPI) is provided on the supercomputer systems. The MPI library is available for C/C++ and Fortran. The library provide blocking and non-blocking functions for sending (uniform and non-uniform) size data and accessing (uniform and non-uniform) size data across ranks through various communication topologies; one-one, one-many,

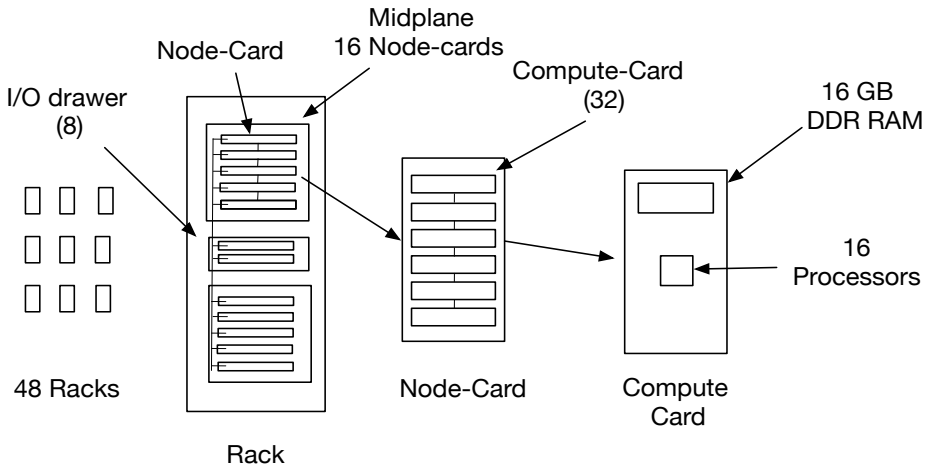


Figure 7: The BG/Q Architecture

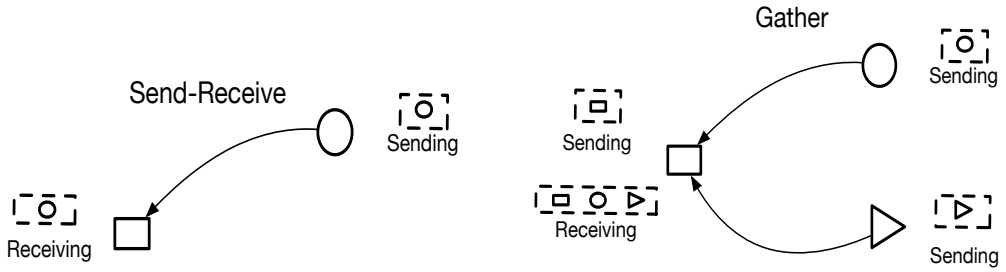
many-one, many-many (AllGather) and many-many (AlltoAll) shown in Figure 8, Figure 9. See (60) for more on MPI documentation.

3.5.1 MPI communication modes

Figure 3.2 and 3.3 below shows the various communication modes within MPI. Each shape represents a rank, the smaller encapsulated shape icons represent the unit of data that the rank is sending or receiving data.

3.6 Job Submissions

All executables must be submitted to a job queue. Job scheduling is provided by the Cobalt job scheduler. There are parameters provided for the job-queue script that allow the user to submit job-time, number of nodes, number of process per node, and alloted time for execution. There are also script provisions to batch jobs together, sequentially after each job completes. Two scripts are used to execute the Mantona job: submit-mantona.py and the start-



(a) (b)

Figure 8: (a) Send-receive: one rank sends the data and one rank receives the data. (b) Gather: one rank is collecting the send data from every rank.

mantona.sh. The submit-mantona.py script is a python script that contains a list of varying Mantona parameters for different jobs, and contains the programming to specify what jobs should be batched together, sequentially being added to the job queue. The start-mantona.sh script is a bash shell script that sets the individual job executable on the queue. An example below of what can be executed from the submit-mantona.py and start-mantona.sh script.

Cetus/Mira/Cooley *submit-Mantona.py*

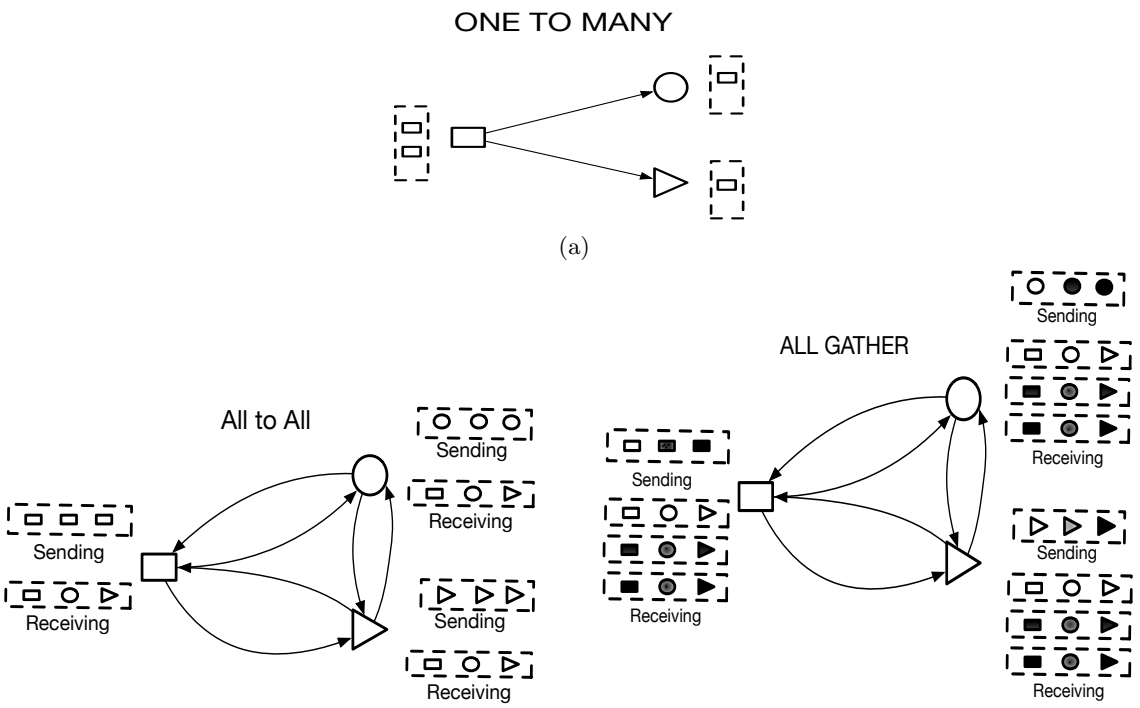
qsub -A *myproject* -t 00:20:00 -n 128 -*mode mantonaScript*

Cetus/Mira *start-mantona.sh*:

runjob -np 16 -p 8 –*block* COLBALT_NAME: *mantona datasetFile* 1

Cooley *start-mantona.sh*:

mpirun -n 128 -f COBALT_NODEFILE *mantona datasetFile* 1



(a) (b) (c)

Figure 9: (a) One-to-many also termed as a broadcast call. The sending rank duplicates its data by the number of ranks and sends it to the other ranks. (b) All-to-all: The sending rank duplicates its data by the number of ranks and sends it to the other ranks. All the ranks receive data from each rank. (c) All-gather: each rank receives $(n * d)^2$ amount of data back in which n is the number of ranks and d is the data size that is to be sent for each rank.

In this example, the job uses 16 nodes with 8 processors per node for a job time of 20 minutes. The executable is *mantonaScript* that loads an rdf dataset *datasetfile*, with a cache depth set to 1. Cetus and Mira executes the program through the *runjob* command and Cooley uses *mpirun*.

3.7 Summary

The research for this thesis was done on a supercomputer in order utilize a large coordinated set of processors to pre-process and process linked RDF queries. To create a high job-throughput supercomputing environment the processing, communication and storage hardware must be efficiently and particularly engineered for high data transfer between the storage unit and processors and among processors. The Argonne supercomputer Mira (and Cetus) is tailored for jobs that require many processors for faster processing of large data at the expense of individual memory while Cooley focuses on a larger memory size per processor at the expense of processor size. Mantona utilizes both types of systems, the former to expedite the pre-processing of the raw RDF data, and the later to to have a larger memory store to fetch the pre-processed data. Mantona utilizes the communication modes of *alltoall*, *onetoall* and *gather* on Mira and Cooley in order to distributively and efficiently pre-process and query the RDF data. Both types of supercomputing environments from Mira and Cooley regarding Mantona pre-processing and query are evaluated and shown in the results chapter.

CHAPTER 4

TERM PROCESSING AND NEIGHBOR NODE GENERATION

Term processing is the transformation of string terms and triples into unique ids (*termList* and *tripleList*) in order to expedite query retrieval processing. Mantona breaks this term processing into two steps. The first step, *reduce terms*, each processor reads its chunk of string terms in the dataset and reduces the terms into a unique list. The next step, *reduce triples*, each processors reads its section of triples in the dataset, replaces the string terms in the triple with ids, then reduces the triples into a unique list. Section 4.1 and 4.2 explains how Mantona uses the collaborative communication modes in MPI to create the *termList* and the *tripleList*. Figure 10 shows the connection between the *tripleList* and *termList*. Another way a query process can be expedited is if each triple can reference its list of connecting triples, *neighborList* and their connecting types. Section 4.3 explains how Mantona uses collaborative communication modes to create the *neighborList*. Section 4.4 explains how terms are collaboratively counted to find the most frequently used terms. The summary is provided in Section 4.5.

4.1 Reduce terms

Each processor reads in a section of terms within the dataset. A dataset, as used in this research, is represented as one file that collects several datasets from the 2016 wiki-DBpedia RDF dataset downloads at http://wiki.dbpedia.org. The MPI-collective read function is used for each core to read a partition of the dataset. Each processor parses through its partition

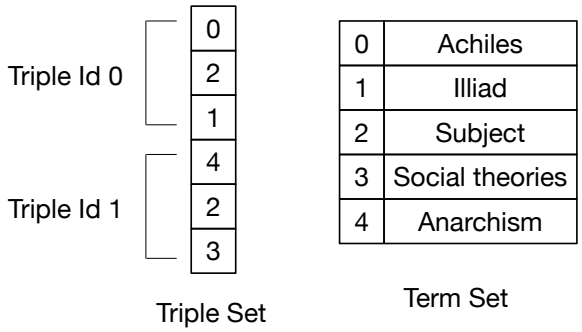


Figure 10: The representation of a triple set (*tripleList*) and a term set (*termList*).

and collects triple terms for each line. For the process of reduce terms, terms are collected individually. Each term is hashed to an index between 0 and n representing the number of ranks. The hash index represents the rank id. The term goes to the processor representing that rank. This reduce process involves and *alltoall* call and an *alltoallv* call. An *alltoall* communication call is used to provide each processor with the number of terms obtained by each processor. The *alltoallv* command sends the terms to the processor according to the term's hashed id. Each processor removes the duplicate terms. Another *alltoall* call is used to send each processor's reduced term count to every processor. The final *alltoallv* communication call provides each processor with the set of reduce terms defined as *termList*. If memory conservation is an issue, then the last *alltoallv* would not be issued and each processor would have its sectional set of reduce terms. The implementation of the graph-cache algorithm and

neighbor generation algorithm would take longer because some triples would have to be fetched from other processors.

4.2 Reduce triples

Each processor reads in a section of triples within the database. The three string terms within a triple are replaced with the term ids taken from the *termList*. Each of the three term ids associated with a triple id are added up and hashed to an index between 0 and *n*. Those three term ids are directed to the processor that represents that hashed id. An *alltoall* call is first invoked to send the triple sizes (the summation of all the terms for each triple) from each processor to every other processor. An alltoallv is then invoked for each processor to send its triples to all processors. Duplicate triples are then removed. An alltoall call is then invoked to send each processor's triple sizes to every processor and the final alltoallv is called to send the triples to every other processor, resulting in the *tripleList*.

4.3 Neighbor generation

Each processor would iterate over each of its root triple. Each root triple would generate the list of triples that have a connection to the root triple. The root triple and its recently generated neighbor list, containing a list of neighbor triples and their connection type would be appended onto an integer array with the value -1 as a delimiter. An *alltoall* call would retrieve the sizes of the integer array for each processor and an *alltoallv* would have each processor receive the integer list from every processor.

4.4 Term counts

It is helpful to know what terms are used most often in order to construct queries with large binding sizes and to get a sense of the dominate subjects and objects of a dataset. Mantona processes the three maximum count subject terms and the three maximum count object terms. In the reduce triple process after the first *alltoall* and *alltoallv* reduce calls, each processor has a unique segment of triples with their corresponding terms. All subject to subject and object to object term counts are created, mapping an individual object and subject term to its term count. The subject and object maps are projected into two individual arrays. The next *alltoall* and *alltoallv* commands will result in each processor receiving the entire subject and object term count arrays. A sort over the term count is then conducted over the subject and object term count arrays. The three highest subject and object term counts are picked and the three lowest subject and object terms counts are picked. These literals are used then to generate patterns containing maximum term subject/object constraints.

4.5 Summary

Term processing is an unavoidable one-time process of transforming an RDF dataset into a unique list of triples, and terms based on integer ids. This type of processing receives little to no evaluation from other related systems researched. The chapter shows how to leverage this process using a large processor based distributed system, in our case, a supercomputer to expedite term processing and the generation of a neighbor list for each triple. The result chapter will show the decrease in term processing and neighbor generation time as the number of processors increase. This chapter also shows how to use processors to effectively generate

maximum and minimum term counts for the subject and object terms. This makes it possible to know the s-s and o-o connections for the maximum and minimum term counts. A *cache-file* can be a very large file (tens to hundreds of gigabytes) depending on the path length specification to store and number of connection types. Choosing a connection type that has a low count, can dramatically reduce the *cache-file* size. Finally, the entire term id and triple id list within Mantona are stored on each processor instead of having it evenly distributed over processors. This will limit the capacity of the pre-processing stages, but will expedite the cache-file processing due to not having to request triple ids or triple terms from different processors.

CHAPTER 5

PATH-CACHE

(PREVIOUSLY PUBLISHED AS LEWIS, MICHAEL J., ET AL. "A DISTRIBUTED GRAPH APPROACH FOR PRE-PROCESSING LINKED RDF DATA USING SUPERCOMPUTERS." PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON SEMANTIC BIG DATA. ACM, 2017 DOI 10.1145/3066911.3066913.)

A path-cache is a tree of linked triples linked triple set, s-s, p-p, o-o, s-o, o-s up to a specific depth that covers the entire combination of linked triples forming a graph. Each linked triple in the set is represented as a path and uses a *signature* to annotate each triple-to-triple connection and *connectionType* to specify how each triple is connected. The linked triple set is created from the triple-roots that are partitioned over the entire set of processors. Each processor saves its portion of the linked triple set into a file, *cache-file*. This chapter explains how the linked triple set is created.

The Mantona preprocessing module is responsible for reading the dataset, transforming an RDF dataset into an RDF graph, identifying neighbor links to each RDF graph node (used for the traversal algorithm, Chapter 6), and for creating a graph in which the nodes represent preprocessed joins. In this chapter, two different techniques are shown for creating a RDF

query system that requires storing all the triples within a processor and one that requires a communication process to access triples throughout the processors.

5.1 Dataset-partitioning

A dataset is represented as one file that collects several datasets from the 2016 wiki-DBpedia RDF dataset downloads at http://wiki.dbpedia.org. The MPI-collective read function is used for each core to read a partition of the dataset. Each processor parses through its partition and collects triple information over each line and collects a list of all the terms.

5.2 Mantona - cache file creation

Mantona name comes from the Sotho word meaning chiefs, where a chief can be viewed as the implemented code within a processor. Each *chief* governs their realm (graph) of linked RDF data. Mantona pre-processes RDF data in the form of paths within the RDF-graph. Mantona first processes an RDF-graph based on s-s,o-s,and s-o links and partitions node assignments to each processor. Each processor generates its own set of sub-graphs term *root-graphs* for each of its assigned *root-ids*. A *root-graph* is composed of nodes termed *path-nodes*. Each *path-node* contains a list of connected triples termed *triple-product* that are generated from the resulting join operations stemming from from the path of intermediate path nodes up to the ending path-node starting from the root node. Figure 11 shows a root-graph from root pattern id *:john* teaches *:English*.

5.3 Path-signature

Each list of triple-products coming from a path node is labeled based on its connection signature. A connection signature is composed of a series of ids that specifies the triples that

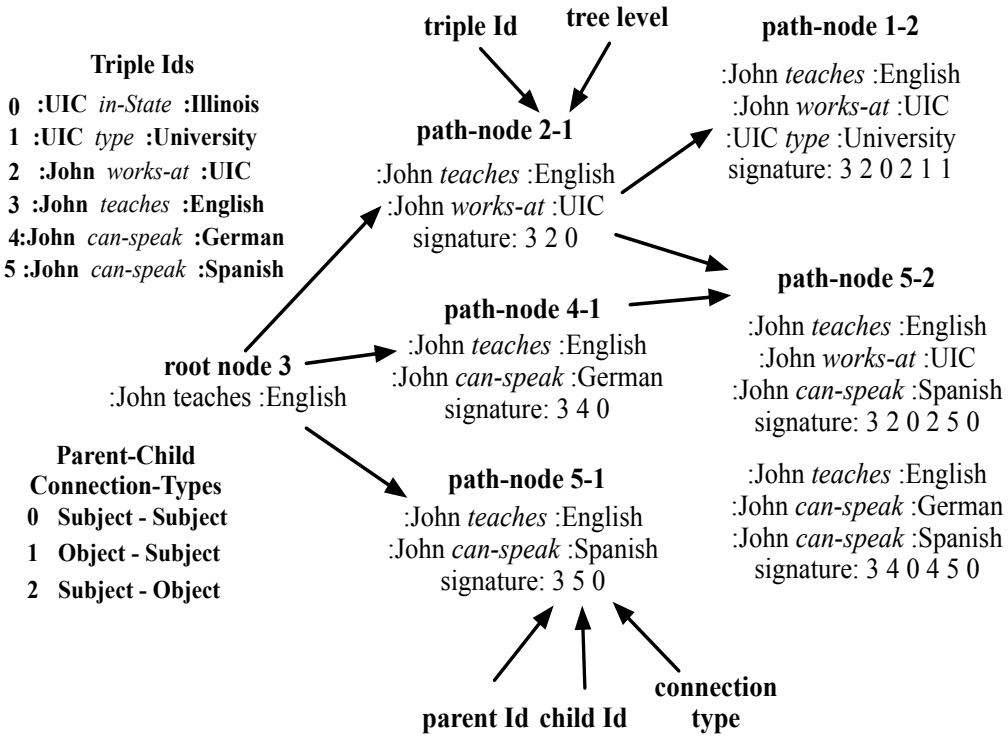


Figure 11: Root-graph from *root-id* 3, using the RDF-graph from Figure 1.

are being connected and its type of link connection: <connected triple id> <in-coming triple id> <connectionType>. A connection type: 0 specifies an s-s connection, 1, o-s connection and 2, s-o connection. The triple-product under path-node 1-2 in Figure 11, has the connection signature *3 2 0 2 1 1*. Root id 3 *:john teaches :english* connects with triple id 2*:john works at :UIC* based on a subject-subject type specification 0. The next connection has id 1 *UIC type University* connecting with id 2 *:john works at :UIC* based on an object-subject specification 1. Path nodes are labeled by the ending connecting triple id and the graph depth. Path-node 1-2 contains all the triple-products of depth 2 that end with id 1 - *:UIC type :University.*

5.4 Mantona Graph-Cache Generation Algorithm

The section shows the root-graph generation algorithm and explains the variables and basic functions within the algorithm. Each processor has a set of root-graphs (*rootGraphList*). For each depth of the growing root-graph the total list (*tripleList*) of triple ids (to be potentially connected to the graph) are checked at the leaf nodes *fringe-nodes*. The *isIn* function determines if there are any term-term connections (s-s, o-s, s-o) between the incoming *id* and the ids within each of the triple products residing within the fringe-node. If there is a connection, a join (*applyJoin*) is made at that connecting triple within the triple-product to create the new linked triple product. The new linked tripled product is added (*insertInPathNode*) to a new path-node. This path-node will become the newest addition to the root-graph and it contains all the the linked triple products of the common ending *id*. All new path-nodes are put on temporary fringe list *addToList*. When all the fringe nodes have been visited, the new path-

nodes become the fringe nodes *swapFringeNodes* and the same procedure continues at the next depth (Algorithm 1).

```
procedure GRAPH-CACHE GENERATOR ;
    for depth ← 1 to maxDepth do
     foreach rootGraph in rootGraphList do
        foreach fringe-node in rootGraph do
            foreach id in tripleList do
                tId = IsIn(id, fringeNode) ;
                if tId > 0 then
                    tp = applyJoin(tId, id) ;
                    insertInPathNode(tp);
                    addToGraph(pathNode);
                    addToList(pathNode);
                end
            end
        end
        swapFringeNodes(pathNode,fringeNode) ;
     end
    end
end procedure
```

Algorithm 1: Graph-Cache Generation Algorithm

5.5 cache-file

The *cache-file* is composed of all the pre-processed elements within the dataset. This includes triple strings, term ids, triple ids, *triple-roots* with their connectivity information (list of neighbor triple ids), and the path-cache data, Figure 12. When reading the cache-file into memory, each processor reads in the termList, tripleList and neighbor list and the tree sizes. Each processor will take a partition of the tree sizes in order to load the corresponding tree size data.

5.6 Summary

The cache-file concept was to create a file which contains pre-processing data: term id, adjacency list of connection information for each triple and path connectivity information or called graph-cache. The file will specify offset information for each type of pre-processing information thus any type of distributing environment that uses MPI can process the cache-file. This chapter presents the path-cache algorithm specifying how connected triple paths can be stored and accessed through a signature.

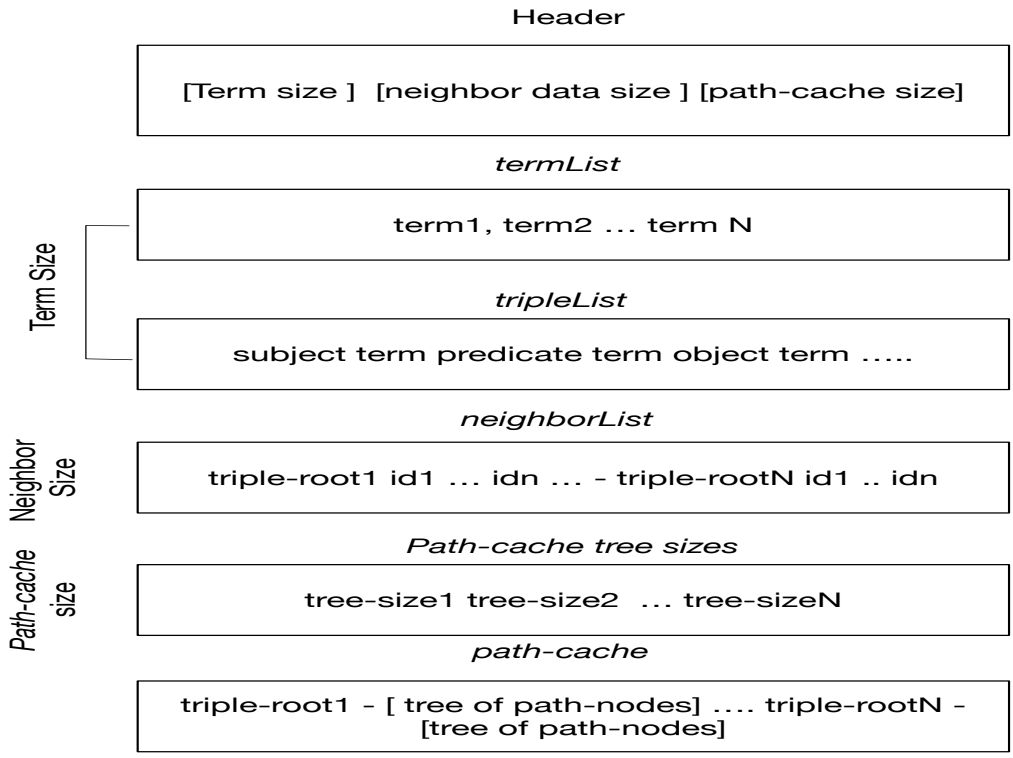


Figure 12: Cache file, containing term,triple, neighbor data and path-cache data.

CHAPTER 6

QUERY INPUT AND RETRIEVAL (PREVIOUSLY PUBLISHED AS LEWIS, MICHAEL J., ET AL. "A DISTRIBUTED GRAPH APPROACH FOR PRE-PROCESSING LINKED RDF DATA USING SUPERCOMPUTERS." PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON SEMANTIC BIG DATA. ACM, 2017 DOI 10.1145/3066911.3066913.)

6.1 Introduction

This chapter explains how a Mantona starts to process queries through two types of input (*random-path* and *term-based*). This chapter explains how input is generated, the steps in retrieving a query using the *path-cache* and the *graph-exploration* algorithm and how the query retrieval algorithms are computationally evaluated.

6.2 Query input generator

Mantona uses two types of input generators to evaluate queries. The *random-path* input generator creates a query string e.g Figure 14, that returns at least one connected query result. The second type of query input generator called *term-based* generator uses the most and least frequently used terms from a dataset. This type of input generator is used to get high or low triple counts per pattern. A high triple count per pattern does not guarantee a high triple result

but it increases the computation in any join that the pattern takes a part in. For both query types used in this work require a binding variable for each pattern, and a binding variable that must only connect to any one of the previously generated patterns.

6.3 *Random Path* Input Generator

In the Random Path Input Generator algorithm, in function *generateAllowedList*, rank 0 randomly picks n (breadth size) processors to be allowed to generate a query. Rank 0 then broadcast out the list to all the ranks so each rank can set the *isAllowed* variable accordingly. Each allowed processor randomly picks a triple from its assigned *rootList* to start its query generation. A loop is iterated up to a *depth* query length. Within the loop a function *connectTriple* looks at the last triple added to the query list and finds a triple that can bind of type *bindingType* (s-s,o-s,s-o,p-p) to the last triple and adds the new triple *inTriple* to the list. The query list is an ordered list of triple ids. The list size or depth is equal to the current iteration count of the algorithm. The order of the list matches the query pattern order. Once the *queryList* is of size depth +1, the query string is created from the query list from the function *generateString*. The parameter *patternType* represents an index value for all Mantona supported patterns. For example a query list: 0 - 1 with pattern type 0 (all subject-binding,

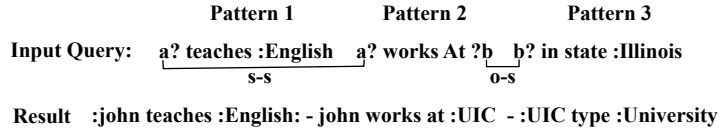


Figure 13: A sample input string of depth 3.

```
procedure RANDOM INPUT GENERATOR(rank,patternType) ;
    allowedList = generateAllowedList(breadth) ;
    if (isAllowed == true) then
     startTriple = rand() % partitionList ;
     queryList.add(startTriple) ;
     for index = 1; index <= depth; index = index + 1 do
         inTriple = connectTriple(queryList.last,bindingType) ;
         if (inTriple == −1) then
             foundConnection = false ;
             break ;
         end
         queryList.add(inTriple)
     end
     if queryList == depth + 1 then
         queryString = generateString(queryList,patternType) ;
     end
    end
    alltoall(queryString.size()) ;
    alltoallv(queryString) ;
end procedure
```

Algorithm 2: Random Input Generation Algorithm

no blank terms), the query string becomes a? in-state :illinois * a? type :University * a?. All processors that have generated a queryString will send the string out to every processor: *alltoall*,*alltoallv*. Each processor will then process the string of connected query patterns using the graph cache algorithm and the traversal algorithm.

6.4 Term Based Input Generator

With the *term-based* input generator, Mantona is able to translate conjunctive SPARQL queries into series of integer based term signatures that can be processed into the graph-cache and graph-exploration retrieval algorithm. Each query pattern is associated with nine integers

grouped in three per term within a pattern. Within a group of three, the first integer represents the pattern index that this term is connected to, the second integer represents the binding type between the terms of the source and target patterns 0: s-s, 1: o-s, 2: s-o, 3: o-o, and the last integer of the group represents the term id of the source pattern. Figure 14 shows a sample SPARQL query with its corresponding term and string mappings. Figure 15 shows the query broken into a 6 pattern input signature.

```
Select *
Where {
 ?a dbpedia:spouse ?b.
 ?a dbpedia:wikilink dbpediares:actor.
 ?b dbpedia:wikilink ?c.
 ?c owl:sameas ?c2.
 ?c2 pos:lat [].
 ?c1 pos:long [].
}
```

Term Id	string
0	Eiffel Tower
1	geo:ontology#inCountry
2	geo:countries#FR
3	pos:lat
4	pos:long
5	dbpedia:spouse
6	dbpedia:wikilink
7	dbpediares:actor
8	dbpediares:placeOfBirth
9	owl:sameAs

Figure 14: Term id table and sample SPARQL input query.

6.4.1 Node-Traversal Algorithm

Mantona has a node traversal algorithm that traverses through all the paths that are represented in a linked query and returns the results only from the matched paths. This is a recursive algorithm, starting at the root-id from *MatchedGraphList*, in which the root-id matches the first pattern within the query pattern. The root-id is inserted in a list of triple products *tpList* and sent to *traversePath(depth,tpList)*. At each call, the depth is checked to see if it is at *maxDepth*. If so the resultant output (triple matches) is printed out, otherwise the traversal algorithm continues to expand the set of triple products (like newly grown branches of a tree) *newTpList* that match with the query pattern at the current depth. The list of neighbors are retrieved from the last id of triple product which represents the previous depth. The *generatetps* function generates a set of triple products resulting from the join of the neighbor id to any of the ids within the triple product (Algorithm 3).

Query processing starts with each processor taking a query string(s) from the random query generator. Mantona parses this string to produce the list of query patterns at each depth and determines the bounded and unbounded terms in each of the patterns. Each process finds if their root-ids match the first query pattern. *MatchedRootGraphs* represents all *rootGraphs* that have the matching *rootId*. A *rootGraph* represents all triple paths that stem from the *rootId*.

The *getNodes* function retrieves all qualifying path-nodes at the *queryDepth* level. So if the the input string consists of 5 linked patterns , Mantona will check all the path-nodes at tree level 4 , and will only accept the path-nodes where its ending connected triple id matches the 5th pattern. Mantona iterates over all the triple-products *tp* within the path-node(s) and compares

```
procedure NODE TRAVERSAL ALGORITHM ;
    foreach rootId in matchedGraphList do
     tp = generate(rootId) ;
     insert(tp, tpList) ;
     traversePath(1, tpList)
    end
end procedure
procedure TRAVERSEPATH(depth, tpList) ;
    if  depth == queryDepth then
     printResult(pathNodes) ;
     return ;
    end
    instantiate(newtpList) ;
    foreach tp in tpList do
     foreach neighbor from tp[depth − 1] do
         generatetps(neighbor, tp, newtpList) ;
     end
    end
    deletetpList ;
    traversePath(depth+1,newtpList) ;
end procedure
```

Algorithm 3: Mantona Node Traversal (Graph Exploration) Algorithm

	?a	dbpedia:spouse	?b
Pattern 0	**-1 -1 -1**	**-1 -1 5**	**-1 -1 -1**
	?a (s-s)	dbpedia:wikilink	dbpediares:actor
Pattern 1	**0 0 -1**	**-1 -1 6**	**-1 -1 7**
	?b (o-s)	dbpedia:wikilink	dbpediares:actor
Pattern 2	**0 1 -1**	**-1 -1 6**	**-1 -1 7**
	?b (o-s)	dbpedia:wikilink	?c
Pattern 3	**0 0 -1**	**-1 -1 6**	**-1 -1 -1**
	?c (o-s)	owl:sameas	?c2
Pattern 4	**3 1 -1**	**-1 -1 9**	**-1 -1 -1**
	?c2 (o-s)	pos:lat	[]
Pattern 5	**4 1 -1**	**-1 -1 3**	**-1 -1 -1**
	?c2 (o-s)	pos:long	[]
Pattern 6	**4 1 -1**	**-1 -1 4**	**-1 -1 -1**

Figure 15: Term input signature.

each connecting triple id and link type to the correlating pattern. If the triple product matches all the patterns in the query in the right order, then its results are printed out (Algorithm 4).

6.5 Graph-Cache Analysis

This section uses the following variables to represent the triple matching formulas for the graph cache and traversal algorithm.

D Depth of the graph.

N Number of triples

```
procedure GRAPH RETRIEVAL ;
    foreach rootGraph in MatchedRootGraphs do
     pathNodeList = getNodes(queryDepth) ;
     foreach pathNode in pathNodeList do
        foreach tripleProduct in pathNode do
            matchingProduct = true;
            foreach triple,index in tripleProduct do
                if Triple_id not in pattern[index] then
                    matchingProduct = false ;
                end
                if Triple_type not in pattern[index] then
                    matchingProduct = false ;
                end
            end
            if matchingProduct == true then
                printOutput(tp);
            end
        end
     end
    end
end procedure
```

Algorithm 4: Mantona Graph-Cache Retrieval Algorithm

$T_{i,j}$ The total number of triple products for the ith path-node at depth j. $T_{0,0} = 1$, which is the root triple. This total number is every triple-product of length i, that has the same ending triple connection and starts with the same triple-root.

P_i The number of path-nodes at depth i. P_D are the leaf path-nodes.

M_{i-1} The total number of triple products that match the pattern at level i-1 where $1 <= i <= D$.

P_{node} The number of leaf graph nodes that match the ending query pattern. A graph node is a collection of triple products in which every triple-product has the same ending triple connection.

C The number of triple-triple connections types. Mantona uses s-s, s-o and o-s connections.

6.5.1 Mantona Graph-Cache formula

The number of triple products within an individual graph.

$$T_i = \sum_{j=1}^{P_i} T_{i,j} \times C \times N - 1 \times i$$

.

For each depth there will at most N-1 candidates to access P_i path-nodes because there can not be any duplicate triples within a path-node and every path-node has the same ending triple. With each candidate that can access a path-node, there are C $\times$ i ways of connecting to an individual triple-product of length i. With C representing the number of connections, Figure 17.

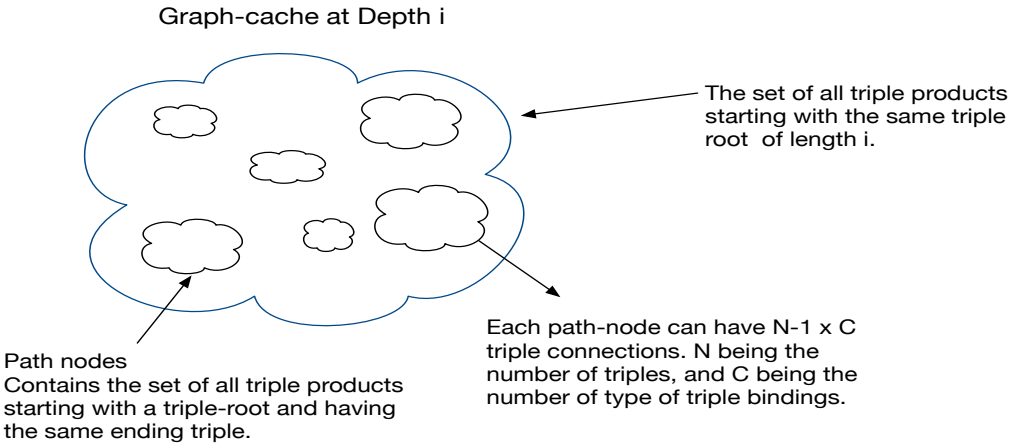


Figure 16: A domain view of Path nodes from the Graph-cache at depth i.

Mantona contains a map of $T_{i,j}$ thus can instantly access path-nodes at any depth. The best search scenario would be to find no match, in which the query pattern string iterates through every path-node at the query depth level. The query string compares the ending triple id for each path node to each triple within the string. If there is not match for any path-node then a not-found match is return. This scenario requires at most N-1 $\times$ query-length comparisons. In (19) the computational complexity is of $O(log_2(n+1))$ in which n is the total number of paths within the graph. In this case the worst scenario would be not to find a match because the whole binary tree would be searched to then find that there was not match.

If there is a match found with the Mantona query search at a processor, each triple-product within the all the matched path nodes is checked. The number of comparisons would be the following in which k is the number of path-nodes that found a match.

$$k \times T_{i,j} \times i \times queryLength$$

6.5.2 Traverse Node-Matching formula

The number of triple-pattern checks for the traverse node algorithm is the following.

$$M_{\mathrm{D}} = \sum_{i=1}^{D} (N) \times C \times M_{\mathrm{i\text{-}1}}$$

At M_0 represents the root node where this triple must match the first pattern within the query. At each level there is a check for each of the N triples to see if the triple matches the pattern and the connectivity type. Faster query results stem from small match sizes resulting from selective queries. Query planning as used in Trinity can provide join configurations using dynamic program where the over all match size can reach a minimum value.

6.6 Summary

This chapter covers the approach of this research in query graph creation, how query retrieval is implemented from the query input and to evaluate the computation complexity of the query processing algorithm. This chapter covered two algorithmic ways to generate a query input. The randomize query input algorithm randomly generates connecting patterns to the growing query input until the algorithm reaches its final connecting pattern. This algorithm was created to

insure that there is at least one result stemming from the query. The other algorithm extracts the most frequent subjects and objects to use as literals for its query. This technique was used, even though it does not guarantee a triple output, the query will contain maximum size patterns. The path-based query can also be used to directly translate the conjunctive section of a SPARQL query into an integer based input that can be read into Mantona. From previous RDF experiments as well as from Mantona experiments, result size is a factor in increasing query timings, thus the aim is to create as much of a result size as possible. The alternative approach with query input is to use the exact queries that were used from related systems. This approach will make it easier to query results across systems and the queries have previously been prepared to fit the user model of queries that produce query results. This chapter also covered the processing of queries through the graph-exploration algorithm and the graph-cache algorithm. The graph-cache is basically a memory based storage structure of categorized paths, in which connected triples can be accessed by keying in the path level that correlates with the query pattern level. This chapter shows how the cache depth, number of triples, the number of connection types, and binding sizes along an accessed path can influence query retrieval times. The graph-exploration algorithm is also shown in recursive programming steps, with each recursive call, the query depth size increases and the tree of paths are created to match the query input at each pattern level. The base condition is reaching the end of the query path or not matching the current pattern. If the former is reached the bindings from the path result is provided. This chapter also shows from a mathematical model how query retrieval timings are directly related to the amount of path-nodes within a level, the number of triple products

within a path-node, the number of connections, and the number of triples. Improvements in the path-cache algorithm would be to traverse the through N triples to see if the visiting triple matches the pattern and if so, then iterate over the all the path-nodes to see if that triple matches the path-node. If the triple matches the path-node, retrieve the paths from the path-node and extract all the paths that match all the patterns up to the current level. The current algorithm iterates over each path-node and checks N triples for a matching current pattern and path-node. The improved algorithm will not have a multiplicative factor of N in its computational time but rather as an additive O(N) to the query retrieval time.

CHAPTER 7

RESULTS

7.1 Introduction

This chapter shows and summarizes the results from the query pre-processing and query timing experiments implemented on Mantona. This chapter also displays and summarizes the query timings experiments extracted from the related query systems: BitMat (4), RDF-3X (3), MonetDB (36), PostgreSQL (61) and Trinty.RDF (17). The query configurations are displayed in the form of a query graph to help the reader map the query binding characteristics to an experimental result. The query graphs show the pattern count, the pattern to pattern connections and the selectivity through the amount, or non presence of blank terms within each pattern. Each line in the query graph represents a pattern. Blank terms represented as '[]' are the non constraint terms. The blank terms that are positioned in the middle of the line are predicate non-constraint terms and the ones at the end of the line segment represent the non-constraining subject/object terms. Each variable shown in the graph represents a join point between two or more patterns. Graph variables are not the same as the variables represented in queries. These graph variables are used to show only the join points in order to better illustrate all of the join connections within the graph. With higher join points, there is a higher chance of increased query timings due to the cross product computations that come from joins.

This chapter contains four sections. Section 7.2 explains the experimental setup for the query trials conducted through RDF-3X, MonetDB, PostgreSQL, BitMat, Trinity.RDF and Map-Reduce and show the results of the experiments. The query graphs are mapped to the bar graphs that show the query retrieval times covering datasets of 9.96 Billion triples, 1.36 billion triples 845 million triples, 21 million triples and 44,114,899 triples. Section 7.3 and 7.4 cover the Mantona experiments. Section 7.3 shows the pre-processing timing results in terms of the time to create and save a *cache-file* for increasing node sizes. This section also compares the time to generate a connectivity list for each triple against building a path-cache. The pre-processing graphs also show the overall time which includes, loading the dataset from file, creating the connectivity list for each triple, creating the path-cache and saving the *cache-file* on file storage. Section 7.4 evaluates query retrieval timings from a term-based input (Q1 - Q6, Figure 23) and a *random-path* input (Q1 - Q3, Figure 25). The term based input trials evaluated the query timing results based on the exploration algorithm and the *path-cache* algorithm over increasing computational nodes using Cooley. The *random-path* query input was used to evaluate query timings from the 500,000, 1,000,000 and 2,000,000 triple dataset on Mira (Figure 26). The *random-path* query input was used to evaluate 100 simultaneous query timings on Cooley from a 250,000 and 500,000 triple dataset (Figure 27). Lastly section 7.5 provides a final summary of the results.

The Mantona jobs were run on Argonne's Cetus, Mira and Cooley systems. Cetus has a 4096 node capacity with a maximum of 16 cores per node and a maximum 1GB per node memory capacity. Cetus has a maximum node capacity of 4096 nodes. Cetus was used to test

the pre-processing experiments with node sizes ranging from 2 nodes, 8 cores to 512 nodes, 2048 cores (4 processors per node). Mira has a 49,152 node capacity with 16 cores per node and a maximum 1GB node memory capacity. Mira was only used to evaluate dataset cache-file using 2048, 4096 and 8192 nodes. Cooley has a 126 node capacity with a maximum of 12 cores per node and maximum of 384 GB RAM per node. Cooley was used to test the pre-processing and query retrieval experiments with core sizes ranging from 2 node 8 processors to 64 node, 256 processors (4 processors per node).

7.2 Results - Related systems

The results shown here are from three sets of experiments using three datasets - LUBM (54) dataset, the Yago (62) dataset and the Uniprot (63) dataset. The query systems involved include MonetDB (36), PostgreSQL (61), RDF-3X (3), and Trinity.RDF (17). These query systems were chosen because they had the fastest query retrieving results available relative to its dataset size and its unique query retrieval techniques. The first three RDF frameworks cover a scan-based, vertical partitioned method to help with query retrieval. The last framework Trinity.RDF uses the graph exploration method to retrieve query results. This section shows the query timings relative to its query graph characteristics.

7.2.0.1 Datasets and Execution Environment

The Yago dataset (62) contains 40,114,899 unique triples and 33,951,636 distinct terms consuming 3.1 GB. Yago consists of data taken from the infoboxes and category system of Wikipedia. The 8 Yago query graphs and query timings for RDF-3X, MonetDB and PostgreSQL are shown in Figure 17. The Yago table data is presented in Table 3. The experiments were

conducted on a Dell D620 PC with a 2Ghz Core 2 Duo Processor with 2 GBytes of memory (3). The cold cache for the Yago experiments were based on dropping the cache from the file systems and starting the queries. The warm cache results were based on running each query five times without dropping the cache and taking the best result.

The Uniprot dataset contains protein data information. It consists of 845,074,885 triples with 147,524,984 subjects, 95 predicates and 128,321,926 objects. The experiments using Uniprot were taken from the BitMat (4) paper, incorporating MonetDB (36), PostgreSQL (61) and RDF-3X (3). BitMat (4) used a Dell 3.0 GhZ, dual processor, 4 GB of memory with 7GB of swap space having 1 TB storage capacity. Results are shown in Figure 18 and Table 4.

The synthetically created Leigh University Benchmark LUBM (54) dataset consists of 9.96 billion triples. This dataset was run on a cluster in which each machine has a 96 Gigbyte DDR3 RAM, two 2.67 Ghz processors, and 6 cores per machine. The experiments were taken from the Trinity.RDF (17) paper. This experiment compared the results from RDF-3X, Map-Reduce-RDF-3X and BitMat. The Map-Reduce-RDF-3X implements the Map-Reduce algorithm using the RDF-3X compression technique. Results are shown in Figure 19 and Figure 20 and Table 5. This paper did not specify how many computational nodes were executing for these trials.

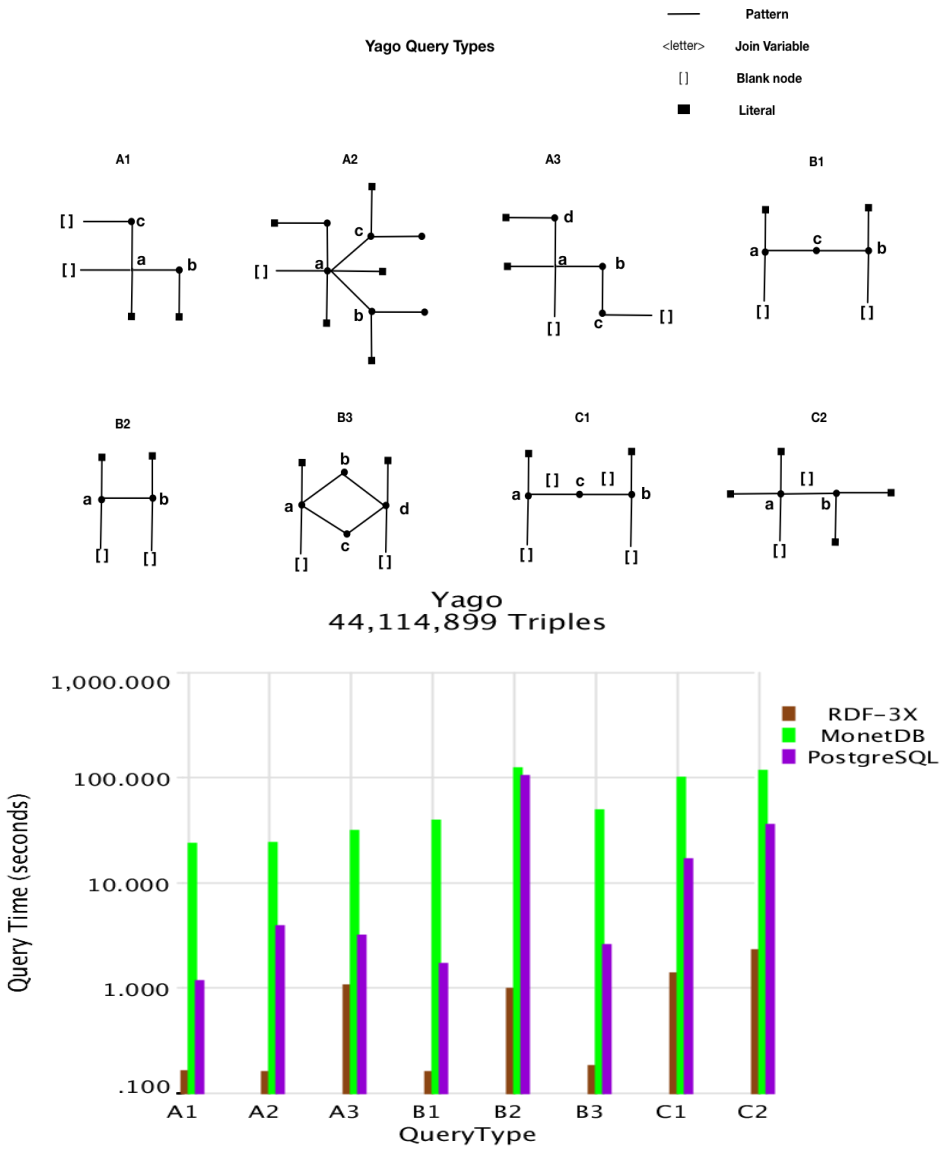


Figure 17: Query timings for 44,114,899 triple dataset covering 8 different query types.

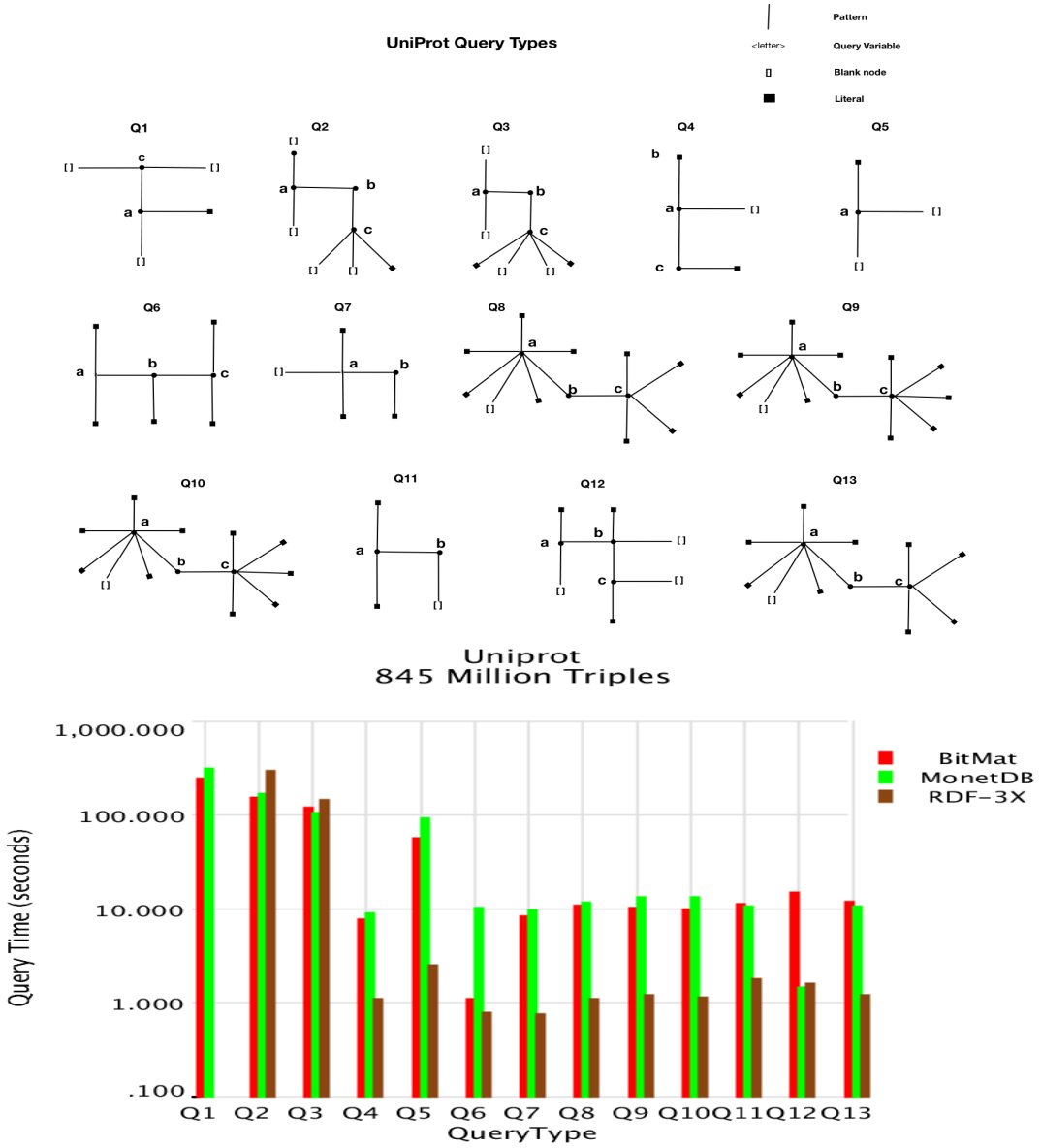


Figure 18: Query timings for the 845 Million triple dataset covering 13 different query types.

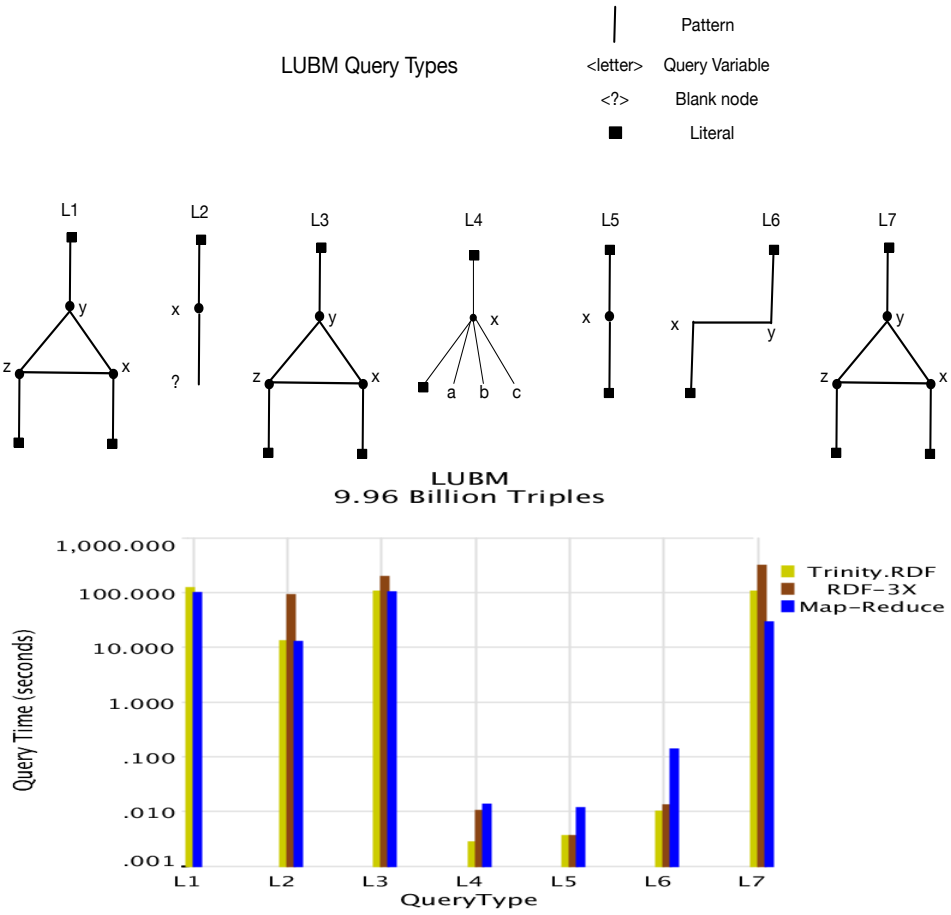


Figure 19: LUBM 9.96 Billion triples.

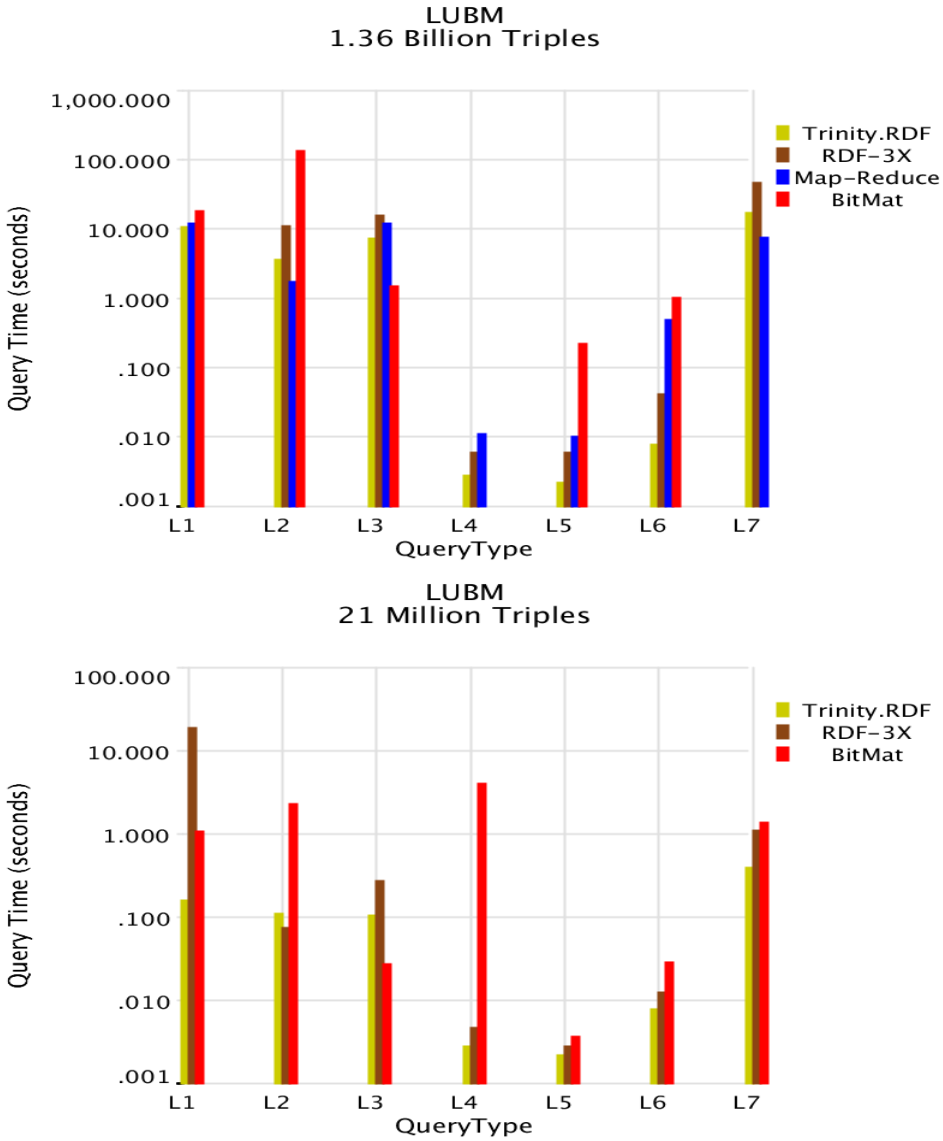


Figure 20: LUBM Top: 1.36 Billion triples as dataset input. Below 21 million triples.

Yago Dataset (Sec)	**A1**	**A2**	**A3**	**B1**
		Cold Cache		
$RDF-3X$	0.29	0.28	1.20	0.28
$MonetDB$	43.55	44.13	54.49	62.94
$PostgreSQL$	1.62	6.31	5.46	3.04
		Warm Cache		
$RDF-3X$	0.02	0.02	0.02	0.01
$MonetDB$	36.92	32.96	34.72	49.95
$PostgreSQL$	0.08	0.43	0.20	0.11
Yago Dataset (Sec)	**B2**	**B3**	**C1**	**C2**
		Cold Cache		
$RDF-3X$	0.99	0.33	2.23	4.23
$MonetDB$	182.39	72.22	101.66	157.11
$PostgreSQL$	117.51	4.71	29.84	59.64
		Warm Cache		
$RDF-3X$	0.05	0.01	0.61	1.44
$MonetDB$	64.84	52.22	84.41	131.35
$PostgresSQL$	7.33	0.12	0.31	50.37

TABLE III: Query run times in seconds for the Yago dataset.

Uniprot 845M	$Q1$	$Q2$	$Q3$	$Q4$	$Q5$	$Q6$
			Cold Cache			
$BitMat$	451.365	269.526	173.324	9.396	78.350	1.340
$MonetDB$	548.210	303.213	124.356	9.630	97.280	11.280
$RDF3X$	Aborted	525.105	244.580	1.380	4.636	0.902
			Warm Cache			
$BitMat$	440.868	263.071	168.673	8.305	77.442	0.448
$MonetDB$	495.640	267.532	113.818	0.584	96.020	0.822
$RDF3X$	Aborted	487.181	226.050	0.077	1.008	0.006

Uniprot 845M	$Q7$	$Q8$	$Q9$	$Q10$	$Q11$	$Q12$	$Q13$
				Cold Cache			
$BitMat$	9.330	13.060	11.430	10.490	15.560	26.980	17.370
$MonetDB$	9.910	15.930	21.370	21.390	12.330	2.468	12.884
$RDF3X$	0.892	1.353	1.718	1.549	3.268	2.804	1.765
				Warm Cache			
$BitMat$	8.360	10.870	9.780	8.690	14.130	25.190	15.770
$MonetDB$	0.861	0.362	0.611	0.563	0.710	0.744	1.020
$RDF3X$	0.003	0.029	0.047	0.046	0.547	0.295	0.046

TABLE IV: Query run-time in milliseconds on the Uniprot dataset (845 Million triples). Table data taken from Matrix "Bit" loaded paper.

LUBM 21M	$L1$	$L2$	$L3$	$L4$	$L5$	$L6$	$L7$
$TrinityRDF$	281	132	110	5	4	9	630
$RDF3X(InMemory)$	34179	88	485	7	5	18	1310
$BitMat(InMemory)$	1224	4176	49	6381	6	51	2168
$RDF3X(ColdCache)$	35739	653	1196	735	367	340	2089
$BitMat(ColdCache)$	1584	4526	286	6924	57	194	2334

LUBM 1.36B	$L1$	$L2$	$L3$	$L4$	$L5$	$L6$	$L7$
$TrinityRDF$	12648	6018	8735	5	4	9	31214
$RDF3X(InMemory)$	$36m47s$	14194	27245	8	8	65	69560
$MapReduceRDF3X$	17188	3164	16932	14	10	720	8868
$BitMat(InMemory)$	33097	209146	2538	*aborted*	407	1057	*aborted*
$RDF3X(ColdCache)$	$39m2s$	18158	34241	1177	1017	993	98846
$MapReduceRDF3X$	32511	7371	19328	675	770	1834	19968
$BitMat(ColdCache)$	39716	225640	9114	*aborted*	494	2151	*aborted*

LUBM 9.96B	$L1$	$L2$	$L3$	$L4$	$L5$	$L6$	$L7$
$TrinityRDF$	176	21	119	0.005	0.006	0.010	126.00
$RDF3X(InMemory)$	*aborted*	96	363	0.011	0.006	0.021	548.00
$MapReduceRDF3X$	102	19	113	0.022	0.016	0.226	51.98
$RDF3X(ColdCache)$	*aborted*	186	1005	874.000	578.000	981.000	700.00
$MapReduceRDF3X$	171	32	151	1.113	0.749	1.428	89.00

TABLE V: Top: Query run-time in milliseconds on the LUBM-160 dataset (21 Million triples). Middle : query run-time in milliseconds on the LUBM-10240 dataset (1.36 billion triples). Bottom : query run-time in seconds on the LUBM-100000 dataset (9.96 billion triples) Table data taken from Trinity.RDF. List of queries shown in appendix B

7.3 Dataset creation Results

These experiments evaluate the time to create the *cache-file* for a 125,000 and 400,000 triple dataset on Cetus and Cooley (Figure 21, Figure 22 (a) and 22 (b). The graphs shown below show three different timings recorded in the dataset construction. The build time represents the time it took to create the *cache-file*. The neighbor time represents the time to find all the neighbor triples including connection type information for each of the nodes. The total time includes the the entire time to pre-process the dataset. This includes loading the dataset, creating the *termList* and *tripleList*, generating the neighbor adjacency list, creating the path-cache and saving all the pre-processed data into a file. The last graph uses Mira to create the *cache-file* containing 1 million triples, and using 2048, 4096 and 8192 cores (Figure 22 (c)). All of these graphs are recorded in seconds.

	Core sizes				
	32	64	128	256	512
Cetus 125,000 triple Dataset (time: seconds)					
$Cache - buildTime$	318.770	159.527	80.176	40.416	20.193
$Neighbor - buildTime$	19.638	10.745	6.320	4.099	3.020
$Total - buildTime$	364.890	198.720	117.190	75.880	109.459
Cetus 400,000 triple Dataset (time: seconds)					
$Cache - buildTime$	3378.580	1533.650	767.212	384.496	192.165
$Neighbor - buildTime$	182.543	98.090	53.701	31.421	20.140
$Total - buildTime$	3735.040	1760.12	932.580	516.440	384.623

	2	4	8	16	32	64
Cooley 125,000 triple Dataset (time: seconds)						
$Cache - buildTime$	837.707	398.490	196.170	98.366	49.134	24.436
$Neighbor - buildTime$	23.230	13.410	7.639	4.811	3.965	5.140
$Total - buildTime$	967.052	445.700	229.600	129.677	89.308	89.351
Cooley 400,000 triple Dataset (time: seconds)						
$Cache - buildTime$	8601.460	4442.23	2176.09	1140.120	502.470	252.921
$Neighbor - buildTime$	236.513	119.20	65.46	38.454	21.559	15.024
$Total - buildTime$	10004.600	4888.43	2496.21	1140.120	623.749	371.256

	Triple sizes		
	2048	4096	8192
Mira 1,000,000 triple Dataset (time: seconds)			
$Cache - buildTime$	2321	1185	599
$Neighbor - buildTime$	22	14	11

TABLE VI: Top: Cetus load times using 125,000 and 400,000 triple dataset. Middle chart: Cooley load times using 125,000 and 400,000 triple dataset. Final chart: Mira load times with 2048, 4096 and 8192 core sizes.

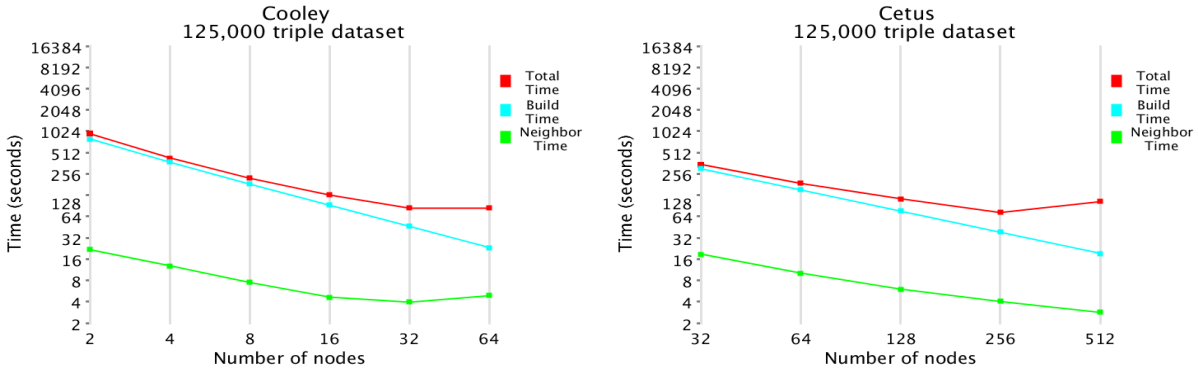


Figure 21: (a) Cache file processing times on a 125,000 triple dataset using Cooley. (b) Cache file processing times on a 125,000 triple dataset using Cetus.

7.4 Query results

The systems Cetus, Cooley and Mira were used to evaluate the query timings for various processor sizes using the path-cache and graph-exploration algorithm. These experiments were set up to examine the effects of query order, results size, and processor size on query timing results.

7.4.1 Cetus and Cooley

Cooley experiments used datasets of triple sizes: 125,000, 400,000 with scaling node sizes of 2,4,8,16,32 and 64, and with 4 processors per node. Cetus used the 50,000 dataset with scaling nodes sizes of 32, 64, 128, 256 and 512 and with 8 processors per node (Figure 24). The 50,000 triple dataset only checked for o-o connections of depth 1 for the *path-cache* and the queries covered Q2-Q6. The *term-based* input was used for these experiments. Six different

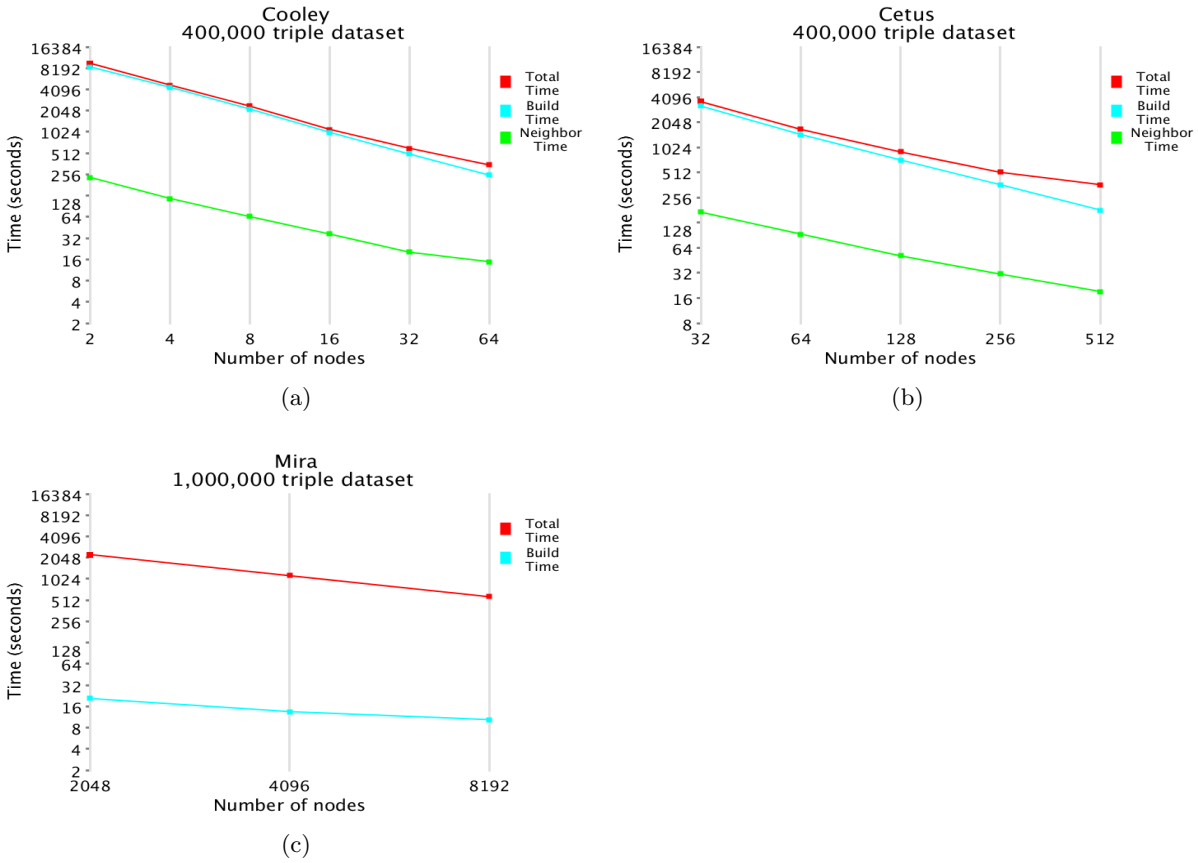


Figure 22: Query results. (a) Cache file processing times on a 400,000 triple dataset using Cooley. (b) Cache file processing times on a 400,000 triple dataset using Cetus. (c) Cache-file processing times on a 1,000,000 triple dataset using Mira.

query configurations (Q1-Q6) were used for the Cooley and Cetus experiments (Figure 24). The literals for these queries are based on the previous processed highest frequencies for the subject and object terms. The description of the queries are : Q1, a two pattern, s-s connection, with one join point and two predicate blank nodes. Q2, a two pattern o-o query with one join point and two predicate blank nodes. Q3, a two pattern o-o connection with one join point and two blank predicate nodes. Q4, Q5 a two pattern o-o query with one join point and two blank nodes on the predicate and subject terms. The difference between queries Q4, Q5 is the pattern processing order. Query Q4 processes the most frequent literal pattern first and query Q5 processes the most frequent literal pattern first. Query Q6, a three pattern o-o query with 1 join points and one blank node. The table data representing the Cetus and Cooley query experiments are shown in Table 7 and Table 8.

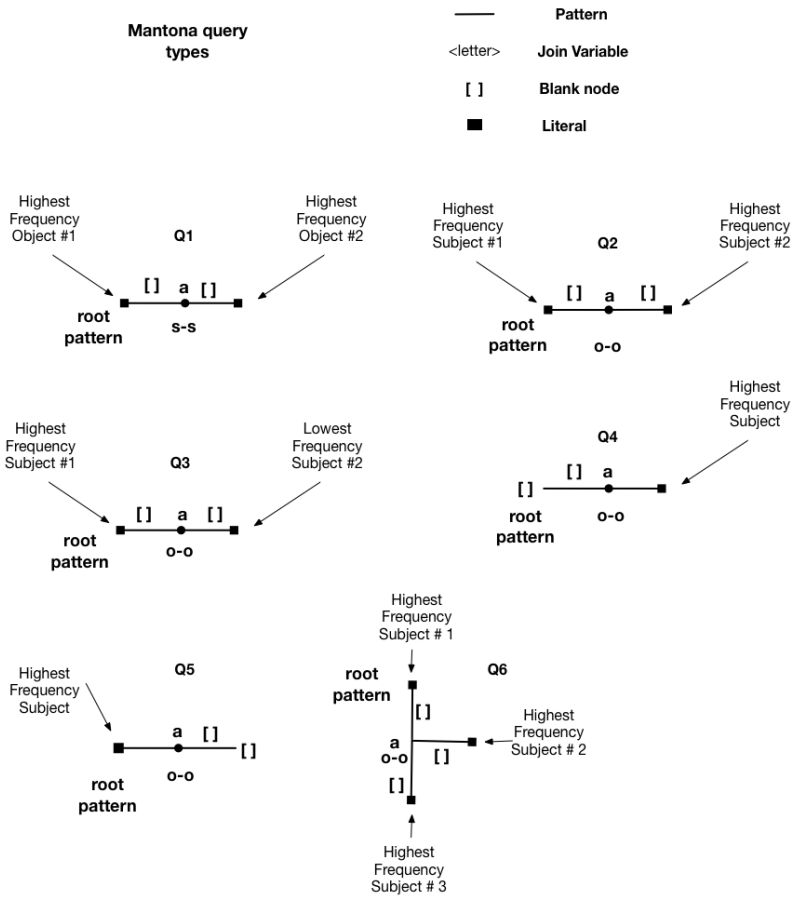


Figure 23: Query graphs Q1 - Q6. Query used for the results shown in Figure 25.

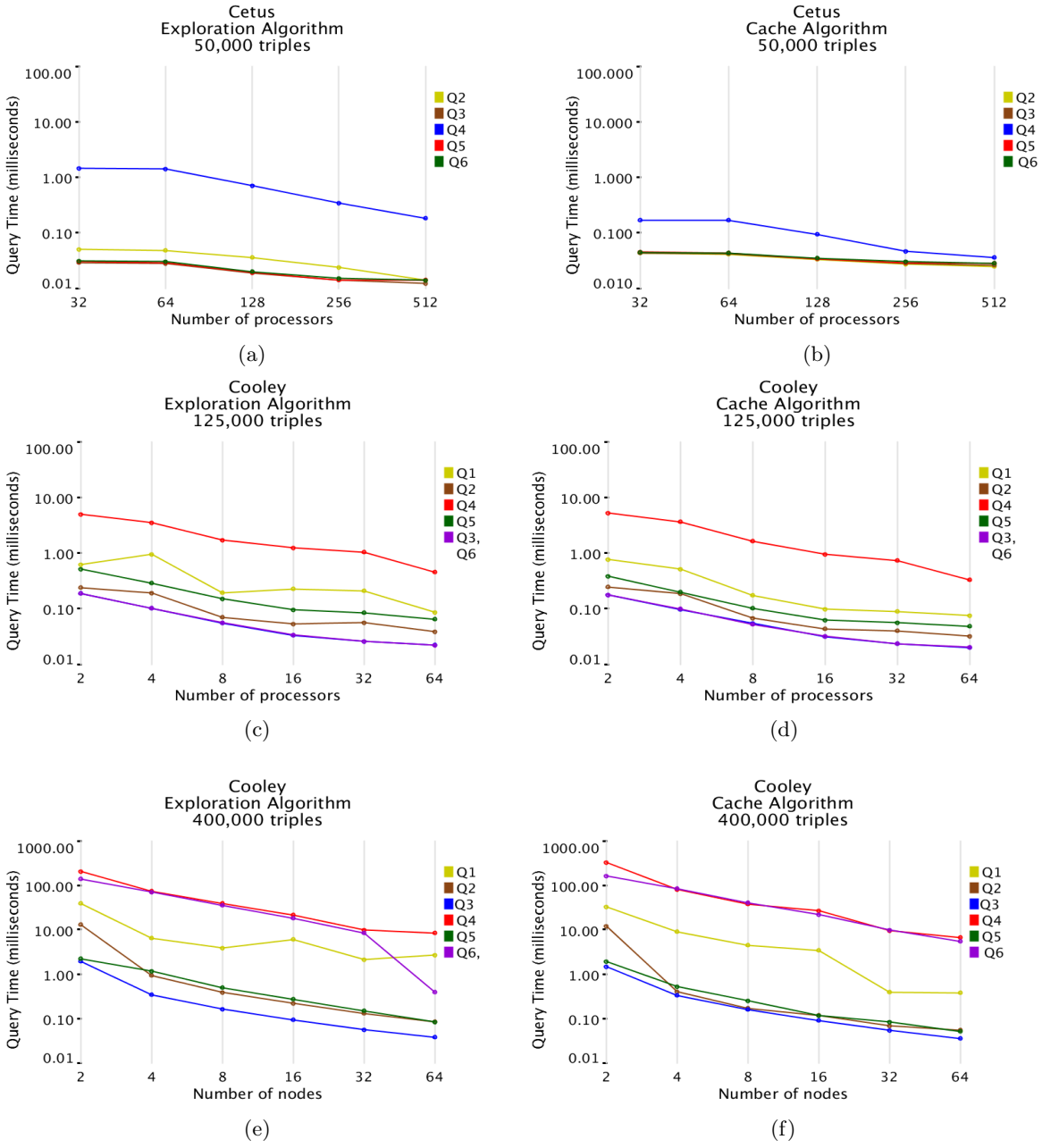


Figure 24: Query timings in milliseconds. (a) Cetus exploration 50,0000 triples. (b) Cetus path-cache 50,000 triples. (c) Cooley exploration 125,000 triples. (d) Cooley path-cache 125,000 triples. (e) Cooley exploration 400,000 triples. (f) Cooley path-cache 400,000 triples.

	Core Sizes					
	2	4	8	16	32	64
50,000 Cooley triple cache algorithm (milliseconds)						
$Q2$	0.120	0.395	0.169	0.118	0.067	0.054
$Q3$	1.460	0.329	0.159	0.089	0.053	0.035
$Q4$	323.000	79.200	37.800	26.700	9.510	6.600
$Q5$	1.900	0.518	0.247	0.115	0.082	0.050
$Q6$	161.000	84.500	40.500	21.800	9.760	5.460
50,000 Cooley triple, exploration algorithm (milliseconds)						
$Q2$	0.530	0.397	0.113	0.080	0.086	0.053
$Q3$	0.390	0.182	0.0841	0.044	0.032	0.026
$Q4$	23.100	15.000	6.120	4.080	3.290	1.170
$Q5$	1.360	0.668	0.294	0.166	0.144	1.170
$Q6$	0.391	0.181	0.853	0.045	0.032	0.026

	32	64	128	256	512
50,000 Cetus triple cache algorithm (milliseconds)					
$Q2$	0.043	0.041	0.033	0.026	0.024
$Q3$	0.042	0.041	0.034	0.028	0.025
$Q4$	0.168	0.169	0.093	0.046	0.036
$Q5$	0.045	0.043	0.033	0.028	0.028
$Q6$	0.044	0.042	0.035	0.030	0.028
50,000 Cetus triple, exploration algorithm (milliseconds)					
$Q2$	0.050	0.047	0.036	0.023	0.013
$Q3$	0.029	0.027	0.018	0.014	0.012
$Q4$	1.430	1.400	0.700	0.341	0.183
$Q5$	0.030	0.029	0.019	0.014	0.014
$Q6$	0.030	0.030	0.020	0.015	0.013

TABLE VII: Query timings, graph-cache and exploration method 50,000 triples.

	Core Sizes					
	2	4	8	16	32	64
	125,000 triple cache algorithm					
$Q1$	32.300	8.970	4.300	3.900	0.387	0.379
$Q2$	0.120	0.395	0.169	0.118	0.067	0.054
$Q3$	1.460	0.329	0.159	0.089	0.053	0.035
$Q4$	323.000	79.200	37.800	26.700	9.510	6.600
$Q5$	1.900	0.518	0.247	0.115	0.082	0.050
$Q6$	161.000	84.500	40.500	21.800	9.760	5.460
	125,000 triple, exploration algorithm					
$Q1$	1.730	2.930	0.404	0.495	0.444	0.146
$Q2$	0.530	0.397	0.113	0.080	0.086	0.053
$Q3$	0.390	0.182	0.084	0.044	0.0326	0.026
$Q4$	23.100	15.000	6.120	4.080	3.290	1.170
$Q5$	1.360	0.668	0.294	0.166	0.144	1.170
$Q6$	0.391	0.181	0.853	0.0452	0.032	0.026
	2	4	8	16	32	64
	400,000 triple cache algorithm					
$Q1$	32.300	8.970	4.430	3.390	0.387	161.000
$Q2$	12.000	0.395	0.169	0.118	0.067	84.500
$Q3$	1.460	0.329	0.159	0.0898	0.053	40.500
$Q4$	323.000	79.200	37.800	26.700	9.510	21.800
$Q5$	1.900	0.518	0.247	0.115	0.082	9.670
$Q6$	161.000	84.500	40.500	21.800	9.760	5.460
	400,000 triple, exploration algorithm					
$Q1$	38.70	6.400	3.810	60.300	2.090	2.660
$Q2$	13.00	0.929	3.810	0.218	0.129	0.0836
$Q3$	1.94	0.340	0.162	0.092	0.055	0.037
$Q4$	202.00	74.000	39.200	21.500	9.740	8.380
$Q5$	2.21	1.160	0.492	269.000	0.148	8.270
$Q6$	139.00	69.600	35.200	18.000	8.43	0.392

TABLE VIII: Query timings, graph-cache and exploration method Cooley, and Cetus, 125,000 and 50,000 triples.

7.4.2 Mira

The Mira experiments used a 500,000, 1,000,000 and 2,000,000 triple dataset over 2048, 4096 and 8192 cores using a *random path* input generator producing query graphs Q0 - Q3, Figure 25. Figure 26 shows the Mira query results using 500,000, 1,000,000 and 2,000,000 triples. These queries were pre-processed to include connected patterns that returned triple results. The description of the queries are: Q0: a s-s connection with one join point, two predicate blank nodes and two literals. Q1: a s-s connection with one join point and two literals. Q2: a s-s connection with one join point and two blank predicate nodes. Q3: a s-s connection with on merge point, and three literal end points. Table 9 shows the results of the Mira experiments explained above. Table 10 shows the result sizes from the Mira experiments. The final experiments, Figure 27 and Table 11 evaluate 100 multiple queries (Q0 and Q1) with 250,000 and 500,000 triples.

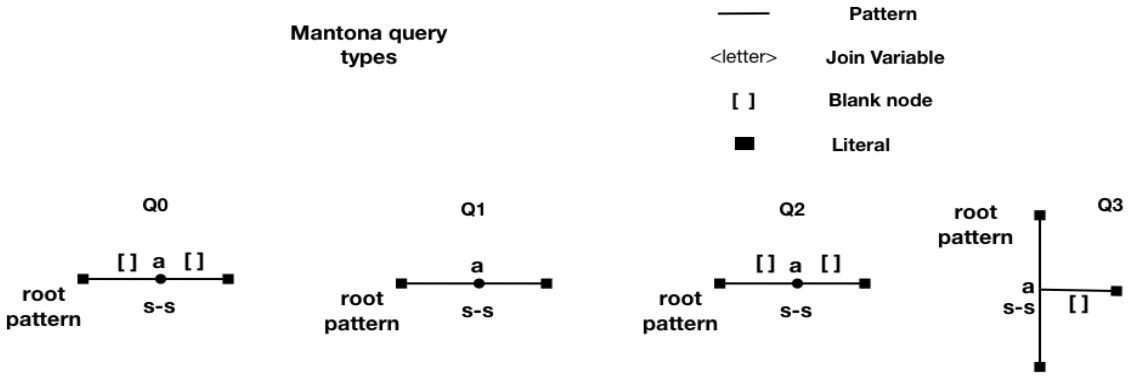
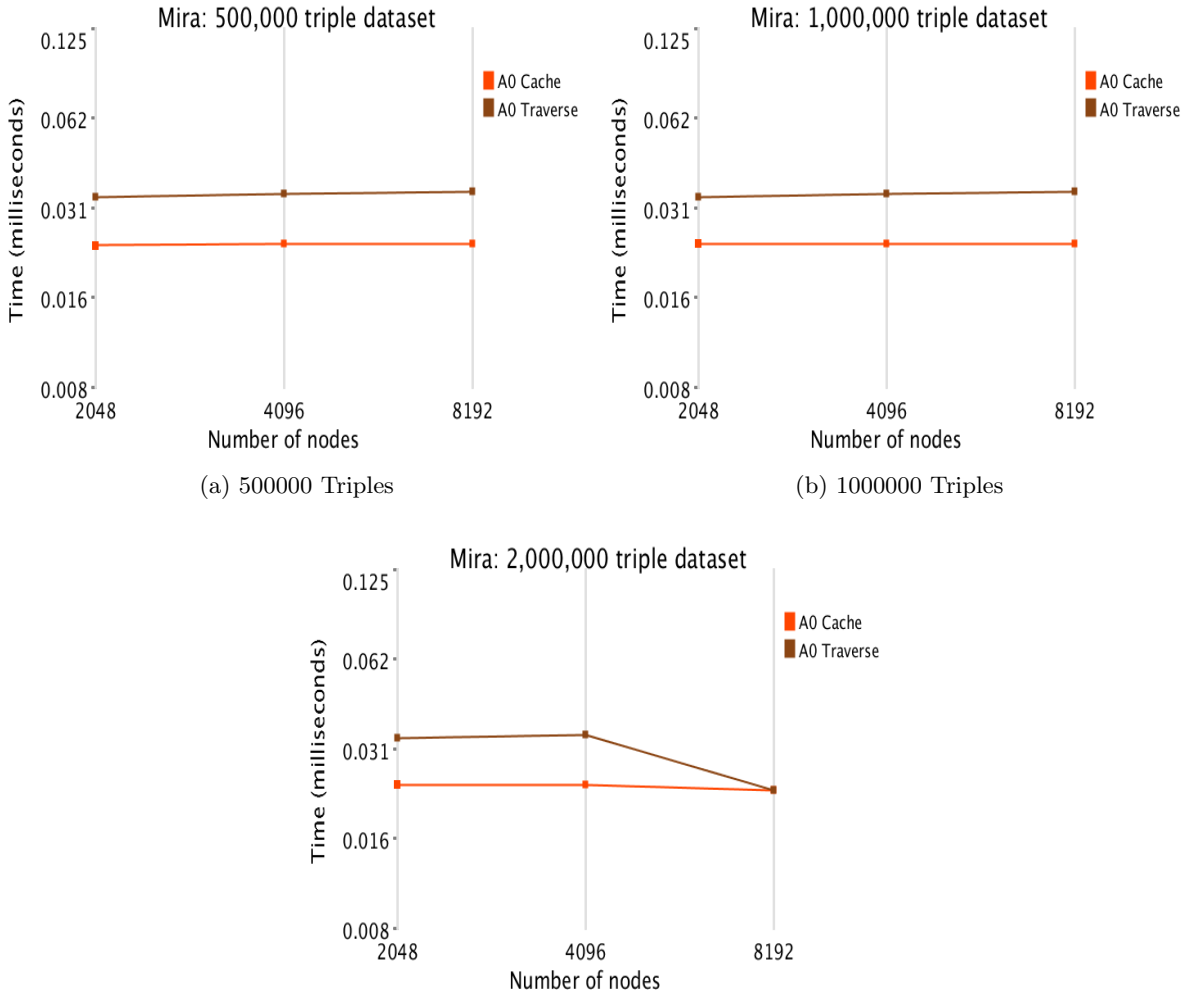


Figure 25: Query graphs Q1 - Q3.



(a) 500000 Triples

(b) 1000000 Triples

Figure 26: Query results (a) .5M triple dataset. (b) 1M triple dataset. (c) 2M triple dataset.

	Core Sizes		
	2048	4096	8192
	500,000 triples		
$Q0 - GraphCache$	24.79	25.03	25.03
$Q0 - GraphTraversal$	35.04	36.00	36.95
$Q1 - GraphCache$	25.03	25.98	25.98
$Q1 - GraphTraversal$	34.80	36.95	36.00
$Q2 - GraphCache$	25.03	25.03	25.98
$Q2 - GraphTraversal$	34.80	36.95	36.00
	1,000,000 triples		
$Q0 - GraphCache$	25.03	25.03	25.03
$Q0 - GraphTraversal$	35.04	36.0	36.0
$Q1 - GraphCache$	26.22	25.98	26.94
$Q1 - GraphTraversal$	35.04	36.00	36.00
$Q2 - GraphCache$	25.03	25.03	25.03
$Q2 - GraphTraversal$	36.0	36.0	36.95
	2,000,000 triples		
$Q0 - GraphCache$	25.03	25.03	24.08
$Q0 - GraphTraversal$	35.04	36.0	24.08
$Q1 - GraphCache$	26.22	25.98	25.98
$Q1 - GraphTraversal$	35.04	36.00	36.00
$Q2 - GraphCache$	25.03	25.03	25.98
$Q2 - GraphTraversal$	36.0	36.0	36.95
$Q3 - GraphCache$	25.03	25.03	25.98
$Q3 - GraphTraversal$	36.00	36.00	36.95

TABLE IX: Query timings on Mira for 500,000, 1,000,000 and 2,000,000 triples over 2048, 4096 and 8192 cores. Time units are in micro-seconds $1 \times 10^{-}6$.

	Result sizes		
	$500,000$	$1,000,000$	$2,000,000$
$Q0$	6	10	10
$Q1$	3	3	9
$Q2$	3	3	9
$Q3$	–	–	9

TABLE X: Result sizes (number of triples) retrieved from the 500,000, 1,000,000 and 2,000,000 triples. Query Q3 was only done on the 2,000,000 triples dataset.

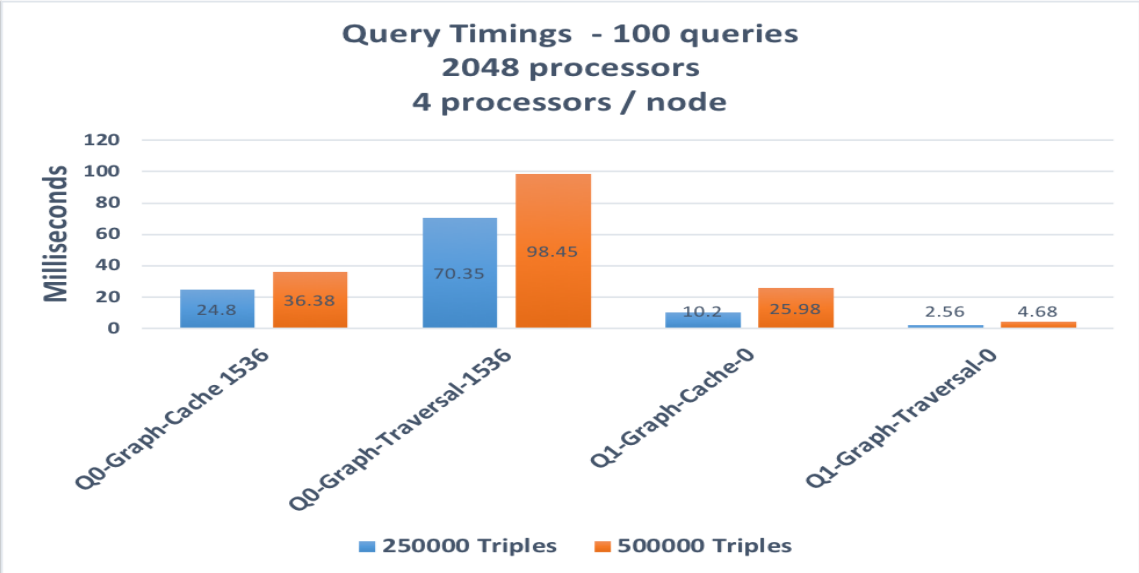


Figure 27: Query timings over 100 simultaneous queries in milliseconds.

	triple size	
	$250,000$	$500,000$
$Q0 - GraphCache - 1536$	24.80	36.38
$Q0 - GraphTraversal - 1536$	70.35	98.45
$Q1 - GraphCache - 0$	10.20	25.98
$Q1 - GraphTraversal - 0$	2.56	4.68

TABLE XI: Time results in milliseconds for one hundred simultaneous queries over 250K and 500K datasets. The number at the end of the query type label is the result size.

7.5 Summary

This chapter evaluates the results from related systems: RDF-3X, MonetDB, PostgreSQL, Trinity.RDF and evaluates the results from Mantona containing the Graph-Exploration algorithm and the graph-cache algorithm.

7.5.1 Related systems

Trinity.RDF (17) has overall the best query retrieval results for larger datasets. Trinity.RDF (17) uses the graph-exploration method for its query retrieval techniques. The Map-Reduce-RDF-3X produces the next best results and like Trinity.RDF it does not have any aborted experiments in its results. With the other systems there is at least one aborted trial over the different datasets and over the different query types. Graph-based algorithms usually are able to scale because the terms or triples and the connectivity information is partitioned over all the compute nodes within the distributed system so one particular compute node can not get overloaded. Also the processor to processor communication infrastructure and the distributed programming library e.g MPI is optimized for fast memory to memory data transfers across processors.

The simulated LUBM query graph L1 matches the query graph L3 but the result sizes are different for the queries which influences timing results. L1 has query results for the 21M and the 1.36B LUBM datasets,but there is no result for L3 query. The bush query (query statements connecting to a single point) L4 has low results for Trinity, RDF-3X, however there is a spike in the query timings for the Map-Reduce - 21 million triples, and the Map-Reduce timings are higher over the other two. Map-reduce can become inefficient when for each iteration there are

just a few more joins happening. This is due to the triple similarities of a bush configuration in which many of the triples are going into the same bucket. RDF-3X however handles bush queries very efficiently.

For scan based algorithms : RDF-3X, MonetDB and PostgreSQL, even using a clustered system the bottleneck comes from variable binding sizes. A particular large pattern size can take more time scanning and selecting the triples from the different tables of vertically partitioned triples scattered over the system. With graph exploration algorithms and map-reduce the collection is dispersed over the processors.

7.5.2 Mantona

The Mantona experiments were set up to evaluate the query pre-processing and query timing processing stages. Due to memory constraints on Cetus/Mira with the entire *tripleList* and *termList* stored on a processor, the experiments were limited to smaller datasets with 2,000,000 computational nodes being the highest that this software can obtain to with its current version. Thus there could be no direct query timing comparisons between the Mantona experiments and the findings from the related systems. Future Mantona revisions will partition the *tripleList* and *termList* over the processors. This will slow down the *path-cache* generation a little but allow Mantona to experiment with much higher datasets.

These Mantona experiments did not match the queries or query graphs to the related systems. Matching a SPARQL query in entirety would be very important in getting accurate comparisons from many SPARQL compliant systems and make this tool a practical tool for users since SPARQL is the main query language used in the RDF community. Mantona ex-

periments using the SPARQL query from the LUBM experiment are evaluated in Chapter 8. From using the *term-base* input, Mantona is able to implement the exact SPARQL queries presented in other query systems with regards to connected queries. Excluding from query graphs, not all queries are the same. For example the SPARQL selection statement *distinct* creates more computation in that all the results must be unique. A map-reduce implementation based on how the *termList* is processed would be necessary to implement this reduction of result terms to its unique set if Mantona was to add the *distinct* feature. Another candidate for the map-reduce process for unimplemented SPARQL features include using the intersection (AND) statement. The SPARQL select feature functions as column/term filtering tool that can be also added to Mantona in future versions in which each processor can filter out the results without required map-reduce functionality. However it is very important for this work to review and evaluate the RDF query results from related systems in order to have a better understanding of the common characteristics of RDF queries and to see what systems and algorithm types show the top query retrieval times. It is also important to show the common rules that other systems operate on which produced the query results and show that Mantona is playing in the same game. Mantona focused on evaluating experiments over the entire life cycle and the relation between a RDF query and an RDF dataset. Mantona separated the entire query process in two stages the pre-process stage and the query process stage. In the pre-process stage, term and triple ids are processed, triple connectivity is processed and graph-cache is generated. In the query retrieval stage the cache-file is retrieved, queries are executed on the system and

the query is processed using some query retrieval algorithm in order to get query timing results.

7.5.3 Query Loading

Query systems do not typically report the pre-processing results, but the pre-processing stage can be time critical as well, even though this is a one time operation. A query user may not have the luxury to wait hours or days to set up the query system with a particular billion plus triple dataset. The results shown in Figure 22 and Figure 23 show that by doubling the number of processors there is a significant reduction (almost by two in some cases) in the pre-processing time. If a system just needs connectivity information the pre-processing time is much smaller (neighbor connectivity information shown in green) than if the system processes the *path-cache* information up to a particular depth (path-cache shown in blue). For a larger node sizes, the results show a smaller reduction rate. More processors makes for a lessor amount of query graph processing per processor. The adding of a large amount of processors does substantially reduce the the pre-processing query timing, but the rate of reduction is much smaller. This may be due to contention with similar data requests from different processors. The total time is the time it takes to process the neighbor connectivity information, the graph-cache processing the term,triple processing. The graph shows that term processing scales as well, when increasing processor size. Also that the term processing is small in comparison to the graph-cache generation time.

7.5.4 Graph-cache vs Graph-exploration

Figure 24 shows query timing results over the graph-cache process (right column) and the graph exploration process (left column). For the small dataset sizes 400,000 and 125,000, the graph cache algorithm does not perform much better than the graph exploration method. This can be due to a small result size or pattern size from the graph query. With a small pattern size the overhead of graph-cache data structures that is being used to retrieve the triples can slow up query processing for smaller datasets.

7.5.5 Query Order

Mantona's query Q4 and Q5 are identical queries but the patterns are processed in different order. Q4 is the graph in red, and Q5 is the graph in green. The query for Q4 and Q5 is to obtain all pairs of triples containing the same object all bounded by the highest frequency subject term. For graph Q4 all of the triples are roots that can then find the o-o pattern that is constrained by the highest frequency term. Query Q4 has all the processors participate in the finding the query since all the triple roots are involved. Query Q5 uses only selected triples roots that have the constraining subject literal, from these roots Mantona looks for any triple that has the o-o connection with the root triple. The data from the graphs show Q4 with the higher retrieval times. This is the case where having more nodes compute the results may be a detriment. Since there are more triples than processors, then there will be a number of triple-roots assigned to each processor. For Q4 the query algorithm will have to be implemented serially for each triple-root, whereas for Q5 there will be a less chance that for a processor to have more than one triple-root that match the constrained subject literal pattern. The next

version of Mantona will need to process query order in way where the most constrained pattern in the query graph should be first used as the root.

CHAPTER 8

LUBM-EXPERIMENT

8.1 Description of Experiment

This experiment used 18,464 triples from the 2.9 GB LUBM-160 dataset. This experiment used Cooley with core sizes 2, 4, 8, 16, 32 and 64. The original dataset was in an XML based RDF format. A program was created to translate the XML based LUBM formated dataset to the n-triple format (64), in which each line contains a triple. This is the format used by Mantona to parse triple datasets. The graph-cache file was created on Cooley using 16 processors taking 81.9 seconds. The *neighborList* (the triple connectivity cache), was processed in 25.2 seconds. Query timings were conducted using the graph-cache algorithm, the graph-exploration algorithm and the *no-cache* exploration algorithm. The *no-cache* algorithm does not use the *neighborList* cache for query retrieval. For the *no-cache* experiment, connectivity had to be calculated on a triple by triple basis. The entire triple list is scanned in order to determine what triples can link up to the source triple.

Here are the following queries used in the LUBM-Experiment.

Q1 SELECT ?x ?y ?z

WHERE {

?z ub:subOrganizationOf ?y .

?y rdf:type ub:University .

```
        ?z rdf:type ub:Department .
        ?x ub:memberOf ?z .
        ?x rdf:type ub:GraduateStudent .
        ?x ub:undergraduateDegreeFrom ?y .
    }
```

Q2 SELECT ?x

```
    WHERE {
        ?x rdf:type ub:Course.
        ?x ub:name ?y .
    }
```

Q3 SELECT ?x ?y ?z

```
    WHERE {
        ?x rdf:type ub:UndergraduateStudent.
        ?y rdf:type ub:University.
        ?z rdf:type ub:Department.
        ?x ub:memberOf ?z.
        ?z ub:subOrganizationOf ?y.
        ?x ub:undergraduateDegreeFrom ?y.
    }
```

Q4 SELECT ?x

```
    WHERE {
        ?x ub:worksFor http://www.Department0.University0.edu.
        ?x rdf:type ub:FullProfessor.
        ?x ub:name ?y1.
        ?x ub:emailAddress ?y2.
```

```
        ?x ub:telephone ?y3.
    }
```

Q5

```
    SELECT ?x ?y
    WHERE {
        ?y ub:subOrganizationOf http://www.University0.edu.
        ?y rdf:type ub:Department.
        ?x ub:worksFor ?y.
        ?x rdf:type ub:FullProfessor.
    }
```

In order to execute each query in Mantona, each term had to be searched for in the term table to obtain its numerical id. Connection type ids had to be applied over each variable, and there was the requirement to specify what triple products will merge with other triple products while remaining compliant with the query. The query transformation into an integer sequence is shown in Figure 16.

8.2 Results

Figure 28 shows the query retrieval times based on Q1-Q5 from the LUBM, 18,464 triple data set. Table 13 shows the results in milliseconds.

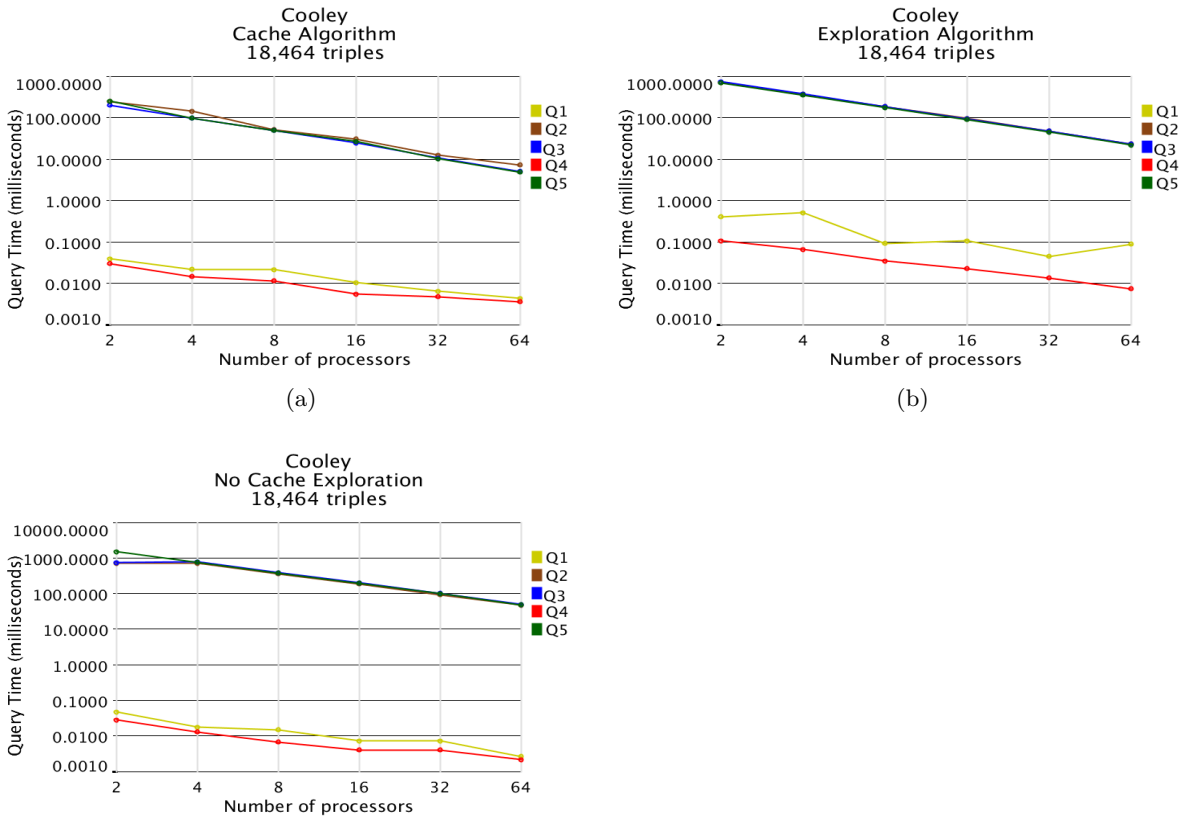


Figure 28: LUBM dataset 18,464 triples. (a) Retrieval times using cache algorithm. (b) Retrieval time using the Cooley exploration algorithm. (c) Retrieval time using the exploration algorithm without using the neighborList cache.

	Core Sizes					
	2	4	8	16	32	64
18,464 Cooley triple cache algorithm (milliseconds)						
$Q1$	0.038	0.021	0.021	0.010	0.006	0.004
$Q2$	243.300	141.140	51.310	30.110	12.330	7.150
$Q3$	198.300	97.075	48.717	24.143	10.619	5.058
$Q4$	0.030	0.014	0.011	0.005	0.004	0.003
$Q5$	246.700	96.600	49.100	26.300	10.200	4.860
18,464 Cooley triple, exploration algorithm (milliseconds)						
$Q1$	0.403	0.499	0.092	0.104	0.044	0.087
$Q2$	710.900	365.685	184.950	94.210	46.730	23.180
$Q3$	734.600	369.900	185.700	94.050	47.530	22.902
$Q4$	0.106	0.064	0.034	0.022	0.013	0.007
$Q5$	692.600	350.700	174.920	88.980	44.700	21.500
18,464 Cooley triple no cache (milliseconds)						
$Q1$	0.046	0.017	0.015	0.007	0.007	0.002
$Q2$	710.940	702.920	358.880	186.630	92.920	46.600
$Q3$	734.600	770.872	387.918	198.977	100.663	49.723
$Q4$	0.027	0.012	0.006	0.004	0.004	0.002
$Q5$	1,480.000	749.800	379.000	194.800	98.100	48.300

TABLE XII: Mantona query timings using the LUBM dataset, 18,464 triples.

LUBM 21M	$Q1$	$Q2$	$Q3$	$Q4$	$Q5$
$TrinityRDF$	281	132	110	5	4
$RDF3X(InMemory)$	34179	88	485	7	5
$BitMat(InMemory)$	1224	4176	49	6381	6
$RDF3X(ColdCache)$	35739	653	1196	735	367
$BitMat(ColdCache)$	1584	4526	286	6924	57

TABLE XIII: LUBM Query timing in milliseconds, 21M triples. Table data taken from Trinity.RDF paper.

8.3 Evaluation of Results

For all 5 different query types the graph cache algorithm performed noticeably better than the exploration algorithm. The differences in query timings were due to binding size and the final triple result size. Q1 and Q4 from its query order had a lower binding size, while Q2, Q3 and Q5 had a higher binding size. As far as scalability in terms of increasing processor size , the experiments show from Table 13 that the query timings decrease as processor size increase. There is no specific ratio of decrease maybe with more experimentation there might be more of a pattern of the rate of decrease, but there is a noticeable difference in general. For example query Q1 went from .038 to .021, 021, .010, 006 and .004 milliseconds as the processor time went from 2, 4, 8, 16, 32, 64. This decrease should continue with processor sizes of 1000, 2000, 5,000 ,10,000 or more for one billion plus triple datasets.

The problem is that there is not enough memory to hold the data in the current form of Mantona, which prevents Mantona from scaling up to the large data sizes. For a billion triples there needs to be at least 120 Gigabytes of memory per processor (for storing the *tripleList* and *termList*) to process datasets of this size. One option is to not store in memory the connectivity list. Figure 29(c) shows query exploration timings that do not use the connectivity cache. These timings are not that different from the exploration algorithm timings. Improvements to Mantona will require the partitioning of the *tripleList* and *termList* and the option to store triple connectivity data. Other improvements would include selective path-cache construction in conjunction with query ordering in order to reduce the memory size per processor.

8.3.1 Bottleneck locations

The LUBM dataset is a very connected dataset. This result highly exposed the bottleneck in Mantona during the pre-processing stage. Within the process of making the cache-file, the *neighborlist* was constructed for each triple root hashed over all the processors. Once the *neighborList* was constructed for each triple-root an MPI alltoallv was executed in order for each processor to receive the local *neighborlist* from every other processor. The *neighborList* consisted of the triple id that was connected to the triple root and the connection id that specified what type of connection this neighbor triple was making, e.g s-s - 0, o-s -1 , s-o - 2, p-p -3 and o-o - 4. Also a neighbor triple can have multiple connection to the same root. For example there can be a subject to subject and predicate to predicate connection binding to the same triple root. This experiment with just 18,464 triples produced 1GB of storage reserved for the *neighborlist*. The other bottleneck came in generating the *termList* and the *tripleList*. The assumption was before the experimentation that query times would always be consistently faster if each processor has the entire list of triples and each containing the entire list of neighbors in memory for lookup, but for small datasets like the current one tested of 18,464 triples, the speed increase was negligible as shown comparing Figure 28 (a) to Figure 28 (c).

8.3.2 How to improve the Mantona bottlenecks

The memory requirement in Mantona is directly connected to the connectivity among triples in the dataset. With the Mantona experiments using the DBpedia datasets, there was not as much triple connectivity and as a result Mantona was able to evaluate datasets ranging up

to 1 million triples. However, those are still small datasets compared to the 1 billion triples evaluated from the other systems shown in this thesis. Eliminating the *neighborlist* processing would free up the connectivity memory bottleneck. The query cost would be based on scanning the entire *tripleList* instead of the smaller *neighborList* when requested. This query timing cost might show more in large billion triple datasets as opposed to smaller datasets.

8.3.2.1 Partition the *termList* and *idList*

Each processor holds the entire *idList* and *termList*. This means that memory size is a direct factor of the number of terms in integer and string representation. So if the average string size of a term is 16 bytes and an integer for a term id is represented as 16 bytes, a billion term dataset with also a billion triples would need to carry 32 GB for the *idList* plus (16 (integer representation) + 48 (number of terms in triple) * 1 billion) = 64 GB for the *termList* representation for a total of 96 GB per processor to minimally handle a billion triple dataset. A way to reduce this excessive memory expense is to partition the *termList* and *tripleList* over all the processors. So , continuing with the example, for 10,000 processors each processor will only have to hold a 960 MB portion of the *tripleList* and *termList*. This means that for the graph exploration algorithm and the graph-cache algorithm every time there is a request for the entire triple list, each processor must iteratively request chunks of the *tripleList* and *termList* from the other processors in order to evaluate connectivity and query matching issues. This can be done through the MPI *alltoallv* call. The adapted graph-cache algorithm and graph-exploration algorithm is shown below.

```
procedure TRAVERSEPATH(depth, tpList) ;
    if depth == queryDepth then
     printResult(pathNodes) ;
     return ;
    end
    instantiate(newtpList) ;
    foreach tripleProduct in tpList do
     foreach triple int tripleProduct do
         red matchList = findMatch(triple) ;
         foreach newTriple in matchList do
             generatetps(neighbor, tp, newtpList) ;
         end
     end
    end
    deletetpList ;
    traversePath(depth+1,newtpList) ;
end procedure
```

Algorithm 5: Mantona Node Traversal (Graph Exploration) Algorithm for scattered triples

The *findMatch* algorithm is a collective algorithm where it would seek triples on other processors that would match the current query pattern and be able to link to the currently selected triple within the query product. These triples that were sought would be sent to the processor that requested it, to be used as a new tripleProduct. The adapted graph-cache algorithm is shown below.

The *findMatch* call would be a collective call within the graph-cache algorithm. It finds the processor that contains the triple and matches the query pattern at pattern[index]. The boolean variable matchingProduct will return true if there is a processor that has the triple and it matches the query pattern, otherwise matchingProduct will return false. A match is based

```
procedure GRAPH RETRIEVAL ;
    foreach rootGraph in MatchedRootGraphs do
     pathNodeList = getNodes(queryDepth) ;
     foreach pathNode in pathNodeList do
         foreach tripleProduct in pathNode do
             matchingProduct = true;
             foreach triple,index in tripleProduct do
                 red matchingProduct = findMatch(triple); ;
             end
             if matchingProduct == true then
                 printOutput(tp);
             end
         end
     end
    end
end procedure
```

Algorithm 6: Mantona Graph-Cache Retrieval Algorithm For Scattered Triples

on if the triple is contained within the current query pattern and if the link connection to the triple product mirror the same link connection of the current query pattern to a previous query pattern.

CHAPTER 9

CONCLUSION

Each area of research in RDF data organization and query retrieval specified in this thesis plays an important complimentary contribution to this field. Data-compression techniques reduce the data-size, thus increasing data loading time and query retrieval time. Applying a map-reduce algorithm to query retrievals enables all active processors to participate in the query retrieval processes when joins are involved. Tree storage data structures, within scan-join systems are beneficial when the retrieved data does not fit entirely in memory. This is typically the case with standalone systems and systems where the amount of data from the dataset overwhelms the number of processing units. Graph based systems utilize graph partitioning algorithms and data communication libraries across processors to reduce query retrieval times. Graph-based systems are able to process queries with scaling dataset sizes that reach billions of triple dataset counts. What this work adds to this field of RDF query processing is the build and utilization of pre-processed connective data within a large distributed environment. Mantona is able to use processor count as a resource for reducing pre-processing and query retrieval timings. The other systems as shown in this thesis do not specify how to use processor size in small and large amounts to affect pre-process and query retrieval timings. Scaling up the cores size in processing a raw RDF dataset has a direct affect in decreasing build times as shown in the experiments. Time can play a factor in dictating if a user would want to use the pre-processing data. Building a *graph-cache* using a large distributed system or a supercomputer

on a particular RDF dataset can provide more useful options to a user. If a system has the processing resources to quickly within seconds build connectivity or even generate *path-cache* data with a limited depth and connectivity, then the process of deleting the old graph-cache, modifying, adding or deleting a triple and rebuilding and saving the *graph-cache* would just take seconds as well, since the core of the processing is the connectivity processing and the path-cache processing. Another factor for/against using a *graph-cache* file is based on the size of the file. Even a 400,000 triple file can produce a gigabyte *cache-file*. Improvements to this work that are critical of storage file limitations can include limiting the connection types allowed for path-cache data. Limited connection types have to be paired with a query order that includes only that connection type. For example if a path cache was created only based on o-o connection types with a depth of one, then any o-o connection between two query patterns must be processed first in order to use the *path-cache*. The rest of the query processing would then be diverted to the query exploration algorithm. From the current experiments, the *path-cache* algorithm is marginally better than the exploration algorithm. More test should be done with larger datasets and larger result sizes to understand how dataset sizes, and binding sizes and path-cache depths can affect path-cache timings. The current path-cache depths were set to one. Further test can be conducted comparing a patch-cache with less connection types and higher depths to path-cache with more connection types but a lower depth. For example, evaluate query timings with an o-o only path-cache with depth 2 to a o-o, s-o, o-s, and s-s *path-cache* with a depth of 1. A user or system can generate multiple queries to have a better understanding of a subject or to help the user make a decision using the given dataset(s). For example if a

user wanted to know if New York or Philadelphia has the best acting schools, the user could first query for all the Oscar winning actors and actresses and link this to names of prominent students that graduated in New York acting schools, and names of prominent students that graduated in Philadelphia acting schools. The user can make a judgment lets say, that New York school is a more conducive place to learn acting because from the multiple queries more Oscar winning actors and actresses went to New York acting schools than Philadelphia schools. This work has shown that a supercomputer can handle multiple queries in a collaborative way and can use processors resource in order to reduce batch query timings.

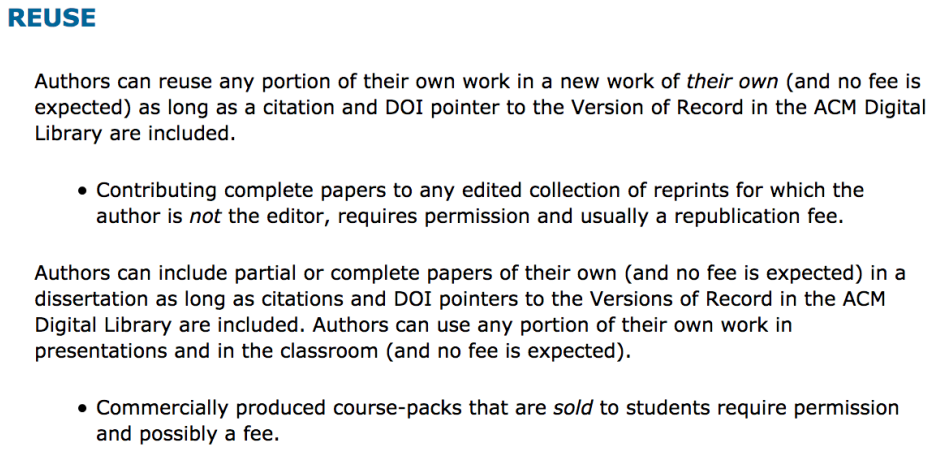
In final summary, Mantona provides the user with a new resource tool (processor size, and using processors in large amounts if the memory is there) in order to reduce query timings whether through individual queries or through batch queries. Mantona also puts the processor size in the equation in order to reduce pre-processing timings by generating a unique *tripleList* and a unique *termList*. Other systems such as BitMat (4), RDF3X (24) and Trinity (17) while able to handle large datasets, does not articulate, or show by experiments the relationship of large processor size (scaling from hundreds to thousands) to the reduction of query timings, batch queries and the time reduction through the pre-processing of triple datasets.

APPENDICES

Appendix A

PERMISSIONS FOR REUSE

Appendix A (Continued)



REUSE

Authors can reuse any portion of their own work in a new work of *their own* (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is *not* the editor, requires permission and usually a republication fee.

Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).

- Commercially produced course-packs that are *sold* to students require permission and possibly a fee.

Figure 29: The permission of reuse for authors published under ACM is specified at : https://authors.acm.org/main.html.

Appendix B

LIST OF QUERIES

```
# Query1
# This query bears large input and high selectivity.
# It queries about just one class and one property.
# Does not assume any hierarchy information or inference.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
    {
        ?X rdf:type ub:GraduateStudent .
        ?X ub:takesCourse .
        http://www.Department0.University0.edu/GraduateCourse0 .
    }


# Query2
# This query increases in complexity.
# 3 classes and 3 properties are involved.
# Also, there is a triangular pattern of relationships
# between the objects involved.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Appendix B (Continued)

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE
    {
        ?X rdf:type ub:GraduateStudent .
        ?Y rdf:type ub:University .
        ?Z rdf:type ub:Department .
        ?X ub:memberOf ?Z .
        ?Z ub:subOrganizationOf ?Y .
        ?X ub:undergraduateDegreeFrom ?Y .
    }

# Query3
# This query is similar to Query 1 but class
# Publication has a wide hierarchy.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE
    {
        ?X rdf:type ub:Publication .
        ?X ub:publicationAuthor
        http://www.Department0.University0.edu/AssistantProfessor0 .
```

Appendix B (Continued)

```
    }


# Query4
# This query has small input and high selectivity.
# It assumes subClassOf relationship between Professor
# and its subclasses.
# Class Professor has a wide hierarchy. Another feature is that
# it queries about multiple properties of a single class.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y1, ?Y2, ?Y3
WHERE {
        ?X rdf:type ub:Professor .
        ?X ub:worksFor <http://www.Department0.University0.edu> .
        ?X ub:name ?Y1 .
        ?X ub:emailAddress ?Y2 .
        ?X ub:telephone ?Y3 .
    }


# Query5
# This query assumes subClassOf relationship between
# Person and its subclasses and
```

Appendix B (Continued)

```
# subPropertyOf relationship between memberOf and its subproperties.
# Moreover, class Person features a deep and wide hierarchy.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
        ?X rdf:type ub:Person .
        ?X ub:memberOf <http://www.Department0.University0.edu> .
    }


# Query6
# This query queries about only one class. But it assumes
# both the explicit subClassOf relationship between
# UndergraduateStudent and Student and the implicit one between
# GraduateStudent and Student. In addition, it has large
# input and low selectivity.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
        ?X rdf:type ub:Student
    }
```

Appendix B (Continued)

```
# Query7
# This query is similar to Query 6 in terms of class
# Student but it increases in the
# number of classes and properties and its selectivity is high.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y
WHERE {
        ?X rdf:type ub:Student .
        ?Y rdf:type ub:Course .
        ?X ub:takesCourse ?Y .
        <http://www.Department0.University0.edu/AssociateProfessor0> \
        ub:teacherOf, ?Y .
    }

# Query8
# This query is further more complex than Query 7 by
# including one more property.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE {
        ?X rdf:type ub:Student .
```

Appendix B (Continued)

```
        ?Y rdf:type ub:Department .
        ?X ub:memberOf ?Y .
        ?Y ub:subOrganizationOf <http://www.University0.edu> .
        ?X ub:emailAddress ?Z .
    }

# Query9
# Besides the aforementioned features of class Student and the
# wide hierarchy of class Faculty, like Query 2.
# This query is characterized by the most classes and properties in
# the query set and there is a triangular pattern of relationships.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y, ?Z
WHERE {
        ?X rdf:type ub:Student .
        ?Y rdf:type ub:Faculty .
        ?Z rdf:type ub:Course .
        ?X ub:advisor ?Y .
        ?Y ub:teacherOf ?Z .
        ?X ub:takesCourse ?Z
    }
```

Appendix B (Continued)

```
# Query10
# This query differs from Query 6, 7, 8 and 9 in that it only requires
# the (implicit) subClassOf relationship between GraduateStudent
# and Student, i.e., subClassOf rela-tionship between
# UndergraduateStudent and Student does not add to the results.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
    ?X rdf:type ub:Student .
    ?X ub:takesCourse
    <http://www.Department0.University0.edu/GraduateCourse0>
    }


# Query11
# Query 11, 12 and 13 are intended to verify the presence of certain
# OWL reasoning capabilities in the system. In this query,
# property subOrganizationOf is defined as transitive. Since in the
# benchmark data, instances of ResearchGroup are stated as a
# sub-organization of a Department individual and the later
# suborganization of a University individual, inference about the
# subOrgnizationOf relationship between instances of ResearchGroup
# and University is required to answer this query.
```

Appendix B (Continued)

```
# Additionally, its input is small.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
        ?X rdf:type ub:ResearchGroup .
        ?X ub:subOrganizationOf <http://www.University0.edu>
    }


# Query12
# The benchmark data do not produce any instances of class Chair.
# Instead, each Department individual is linked to the chair professor
# of that department by property headOf. Hence this query requires
# realization, i.e., inference that that professor is an instance
# of class Chair because he or she is the head of a department.
# Input of this query is small as well.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y
WHERE {
        ?X rdf:type ub:Chair .
        ?Y rdf:type ub:Department .
        ?X ub:worksFor ?Y .
```

Appendix B (Continued)

```
        ?Y ub:subOrganizationOf <http://www.University0.edu>
    }


# Query13
# Property hasAlumnus is defined in the benchmark ontology as the
# inverse of property degreeFrom, which has three subproperties:
# undergraduateDegreeFrom, mastersDegreeFrom, and doctoralDegreeFrom.
# The benchmark data state a person as an alumnus of a university
# using one of these three subproperties instead of hasAlumnus.
# Therefore, this query assumes subPropertyOf relationships between
# degreeFrom and its subproperties,
# and also requires inference about inverseOf.
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE {
        ?X rdf:type ub:Person .
        <http://www.University0.edu> ub:hasAlumnus ?X
    }


# Query14
# This query is the simplest in the test set.
# This query represents those with large input and low selectivity
```

Appendix B (Continued)

```
# and does not assume any hierarchy information or inference.

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>

SELECT ?X

WHERE {

        ?X rdf:type ub:UndergraduateStudent

    }
```

Appendix C

TEST QUERIES

The test queries are listed below. Each query is an input string composed of conjunctive patterns. A pattern is three terms with each term encapsulated in "< >". The variable terms are a string that includes a "?" such as <a?>. Blank terms are only the "?" character. Result are listed below the query.

File: article-500000

Q0:

<a?> <?> <http://dbpedia.org/resource/Category:Harrisburg_metropolitan_area>

<a?><?> <http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

Results

<http://dbpedia.org/resource/Toboyne_Township,_Perry_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Harrisburg_metropolitan_area>

<http://dbpedia.org/resource/Spring_Township,_Perry_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

<http://dbpedia.org/resource/Jackson_Township,_Perry_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Harrisburg_metropolitan_area>

Appendix C (Continued)

<http://dbpedia.org/resource/Middletown,_Dauphin_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

<http://dbpedia.org/resource/Southwest_Madison

_Township,_Perry_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

<http://dbpedia.org/resource/Carroll_Township,_Perry_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

<http://dbpedia.org/resource/NorthEast_Madison

_Township,_Perry_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

<http://dbpedia.org/resource/Marysville,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

<http://dbpedia.org/resource/Saville_Township,_Perry_County,_Pennsylvania>

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

<http://dbpedia.org/resource/Tyrone_Township,_Perry_County,_Pennsylvania>

Appendix C (Continued)

<http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1755>

Q1:

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Pasadena_City_Lancers_baseball_players>

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Burials_at_Cypress_Hills_Cemetery

_(New_York_City)>

Results:

<http://dbpedia.org/resource/Jackie_Robinson> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Pasadena_City_Lancers_baseball_players>

<http://dbpedia.org/resource/Jackie_Robinson> <http://purl.org/dc/terms/subject>

http://dbpedia.org/resource/Category:Burials_at_Cypress_Hills_Cemetery

_(New_York_City)>

Q2:

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:1598_deaths>

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Businesspeople_from_Paris>

Results:

<http://dbpedia.org/resource/Henri_Estienne> <http://purl.org/dc/terms/subject>

Appendix C (Continued)

<http://dbpedia.org/resource/Category:1598_deaths>

<http://dbpedia.org/resource/Henri_Estienne> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Businesspeople_from_Paris>

File: article-1000000

Q0:

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Municipalities_of_Rogaland>

<a?><http://purl.org/dc/terms/subject> <http://dbpedia.org/resource/Category:Kvitsy>

Result

<http://dbpedia.org/resource/Category:Kvitsy> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Municipalities_of_Rogaland>

<http://dbpedia.org/resource/Category:Kvitsy> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Kvitsy>

Q1:

<a?><?><http://dbpedia.org/resource/Category:Fermentation_in_food_processing> <a?>

<?> <http://dbpedia.org/resource/Category:Edible_fungi>

Results:

<http://dbpedia.org/resource/Baker's_yeast> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Fermentation_in_food_processing>

<http://dbpedia.org/resource/Baker's_yeast> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Edible_fungi>

Appendix C (Continued)

Q2

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Lake_Nasser>

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Nefertari>

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:World_Heritage_Sites_in_Egypt>

Result

http://dbpedia.org/resource/Abu_Simbel_temples http://purl.org/dc/terms/subject

http://dbpedia.org/resource/Category:Lake_Nasser

http://dbpedia.org/resource/Abu_Simbel_temples http://purl.org/dc/terms/subject

http://dbpedia.org/resource/Category:Nefertari

http://dbpedia.org/resource/Abu_Simbel_temples http://purl.org/dc/terms/subject

http://dbpedia.org/resource/Category:World_Heritage_Sites_in_Egypt

Q3

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:American_pro-life_activists>

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:War_Resisters_League_activists>

<a?> <?> <http://dbpedia.org/resource/Category:Jazz_writers>

Result

Appendix C (Continued)

<http://dbpedia.org/resource/Nat_Hentoff> <http://purl.org/dc/terms/subject>
<http://dbpedia.org/resource/Category:American_pro-life_activists>

<http://dbpedia.org/resource/Nat_Hentoff> <http://purl.org/dc/terms/subject>
<http://dbpedia.org/resource/Category:American_pro-life_activists>

<http://dbpedia.org/resource/Nat_Hentoff> <http://purl.org/dc/terms/subject>
<http://dbpedia.org/resource/Category:American_pro-life_activists>

File: article-2000000

Q0

<a?><?> <http://dbpedia.org/resource/Category:York>

<a?><?>
<http://dbpedia.org/resource/Category:River_navigations_in_the_United_Kingdom> **Result**

<http://dbpedia.org/resource/River_Foss> <http://purl.org/dc/terms/subject>
<http://dbpedia.org/resource/Category:York>

<http://dbpedia.org/resource/River_Foss><http://purl.org/dc/terms/subject>
http://dbpedia.org/resource/Category:River_navigations_in_the_United_Kingdom>

Q1

<a?><http://purl.org/dc/terms/subject>
<http://dbpedia.org/resource/Category:Petroleum_politics>

<a?><http://purl.org/dc/terms/subject>
<http://dbpedia.org/resource/Category:1973_in_international_relations>

Appendix C (Continued)

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Cold_War_history_of_Japan>

Result

<http://dbpedia.org/resource/1973_oil_crisis> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Petroleum_politics>

<http://dbpedia.org/resource/1973_oil_crisis> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:1973_in_international_relations>

<http://dbpedia.org/resource/1973_oil_crisis> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Cold_War_history_of_Japan>

Q2

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1589 >

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Hero_Cities_of_the_Soviet_Union >

<a?><?><http://dbpedia.org/resource/Category:Populated_places_on_the_Volga > **Result**

<http://dbpedia.org/resource/Volgograd><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1589>

<http://dbpedia.org/resource/Volgograd><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Hero_Cities_of_the_Soviet_Union>

<http://dbpedia.org/resource/Volgograd><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_on_the_Volga>

Appendix C (Continued)

Q3

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:County_seats_in_Indiana>

<a?><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1836>

Result

<http://dbpedia.org/resource/Decatur,_Indiana><http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:County_seats_in_Indiana>

<http://dbpedia.org/resource/Decatur,_Indiana> <http://purl.org/dc/terms/subject>

<http://dbpedia.org/resource/Category:Populated_places_established_in_1836>

CITED LITERATURE

1. Berners-Lee, T., Hendler, J., and Lassila, O.: The semantic web. Scientific american, 284(5):34–43, 2001.

2. Fensel, D.: Spinning the Semantic Web: bringing the World Wide Web to its full potential. Mit Press, 2005.

3. Neumann, T. and Weikum, G.: The rdf-3x engine for scalable management of rdf data. The VLDB JournalThe International Journal on Very Large Data Bases, 19(1):91–113, 2010.

4. Atre, M., Chaoji, V., Zaki, M. J., and Hendler, J. A.: Matrix bit loaded: a scalable lightweight join query processor for rdf data. In Proceedings of the 19th international conference on World wide web, pages 41–50. ACM, 2010.

5. Kolas, D., Emmons, I., and Dean, M.: Efficient linked-list rdf indexing in parliament. SSWS, 9:17–32, 2009.

6. Weiss, C., Karras, P., and Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment, 1(1):1008–1019, 2008.

7. Ladwig, G. and Harth, A.: Cumulusrdf: linked data management on nested key-value stores. In The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011), volume 30, 2011.

8. Aranda-Andújar, A., Bugiotti, F., Camacho-Rodríguez, J., Colazzo, D., Goasdoué, F., Kaoudi, Z., and Manolescu, I.: Amada: web data repositories in the amazon cloud. In Proceedings of the 21st ACM international conference on Information and knowledge management, pages 2749–2751. ACM, 2012.

9. Punnoose, R., Crainiceanu, A., and Rapp, D.: Rya: a scalable rdf triple store for the clouds. In Proceedings of the 1st International Workshop on Cloud Intelligence, page 4. ACM, 2012.

10. Huang, J., Abadi, D. J., and Ren, K.: Scalable sparql querying of large rdf graphs. Proceedings of the VLDB Endowment, 4(11):1123–1134, 2011.

11. Zhang, X., Chen, L., Tong, Y., and Wang, M.: Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud. In 2013 IEEE 29th International Conference on Data Engineering (ICDE), pages 565–576. IEEE, 2013.

12. Huang, J., Abadi, D. J., and Ren, K.: Scalable sparql querying of large rdf graphs. Proceedings of the VLDB Endowment, 4(11):1123–1134, 2011.

13. Galárraga, L., Hose, K., and Schenkel, R.: Partout: a distributed engine for efficient rdf processing. In Proceedings of the 23rd International Conference on World Wide Web, pages 267–268. ACM, 2014.

14. Hose, K. and Schenkel, R.: Warp: Workload-aware replication and partitioning for rdf. In Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on, pages 1–6. IEEE, 2013.

15. Miller, J. J.: Graph database applications and concepts with neo4j. In Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA, volume 2324, page 36, 2013.

16. Cray graph engine: Graph database and graph analytics.

17. Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z.: A distributed graph engine for web scale rdf data. In Proceedings of the VLDB Endowment, volume 6, pages 265–276. VLDB Endowment, 2013.

18. Groppe, S., Groppe, J., and Linnemann, V.: Using an index of precomputed joins in order to speed up sparql processing. In ICEIS (1), pages 13–20, 2007.

19. Matono, A., Amagasa, T., Yoshikawa, M., and Uemura, S.: An indexing scheme for rdf and rdf schema based on suffix arrays. In SWDB, pages 151–168, 2003.

20. Manola, F.: Rdf primer w3c recommendation 10 february 2004. http://www. w3. org/TR/2004/REC-rdf-primer-20040210/, 2007.

21. Owl web ontology language overview, 2004.

22. Date, C. J.: An introduction to database systems. Pearson Education India, 2006.

23. Prud, E., Seaborne, A., et al.: Sparql query language for rdf. 2006.

24. Neumann, T. and Weikum, G.: The rdf-3x engine for scalable management of rdf data. The VLDB JournalThe International Journal on Very Large Data Bases, 19(1):91–113, 2010.

25. Sen, R., Farris, A., and Guerra, P.: Benchmarking apache accumulo bigdata distributed table store using its continuous test suite. In Big Data (BigData Congress), 2013 IEEE International Congress on, pages 334–341. IEEE, 2013.

26. Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al.: C-store: a column-oriented dbms. In Proceedings of the 31st international conference on Very large data bases, pages 553–564. VLDB Endowment, 2005.

27. Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., and Bear, C.: The vertica analytic database: C-store 7 years later. Proceedings of the VLDB Endowment, 5(12):1790–1801, 2012.

28. Du, J.-H., Wang, H.-F., Ni, Y., and Yu, Y.: Hadooprdf: A scalable semantic data analytical engine. In International Conference on Intelligent Computing, pages 633–641. Springer, 2012.

29. Papailiou, N., Tsoumakos, D., Konstantinou, I., Karras, P., and Koziris, N.: H 2 rdf+: an efficient data management system for big rdf graphs. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pages 909–912. ACM, 2014.

30. Kim, H., Ravindra, P., and Anyanwu, K.: From sparql to mapreduce: The journey using a nested triplegroup algebra. Proc. VLDB Endow, 4(12):1426–1429, 2011.

31. Ravindra, P., Kim, H., and Anyanwu, K.: An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. In Extended Semantic Web Conference, pages 46–61. Springer, 2011.

32. Yamamoto, Y.: On indices for xml documents with namespaces. In Conference Proc. Markup Technologies' 99, GCA, Philadelphia, USA, 1999, 1999.

33. Udrea, O., Pugliese, A., and Subrahmanian, V.: Grin: A graph based rdf index. In AAAI, volume 1, pages 1465–1470, 2007.

34. Wilkinson, K. and Wilkinson, K.: Jena property table implementation, 2006.

35. Broekstra, J., Kampman, A., and Van Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In International semantic web conference, pages 54–68. Springer, 2002.

36. Boncz, P., Grust, T., Van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J.: Monetdb/xquery: a fast xquery processor powered by a relational engine. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pages 479–490. ACM, 2006.

37. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2):4, 2008.

38. Guide, D.: Amazon simple storage service. 2008.

39. Lakshman, A. and Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

40. Karypis, G. and Kumar, V.: Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

41. Afrati, F. N. and Ullman, J. D.: Optimizing multiway joins in a map-reduce environment. IEEE Transactions on Knowledge and Data Engineering, 23(9):1282–1298, 2011.

42. Afrati, F. N. and Ullman, J. D.: Optimizing joins in a map-reduce environment. In Proceedings of the 13th International Conference on Extending Database Technology, pages 99–110. ACM, 2010.

43. Blanas, S., Patel, J. M., Ercegovac, V., Rao, J., Shekita, E. J., and Tian, Y.: A comparison of join algorithms for log processing in mapreduce. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 975–986. ACM, 2010.

44. Doulkeridis, C. and NØrvåg, K.: A survey of large-scale analytical query processing in mapreduce. The VLDB JournalThe International Journal on Very Large Data Bases, 23(3):355–380, 2014.

45.

46. Schätzle, A., Przyjaciel-Zablocki, M., and Lausen, G.: Pigsparql: Mapping sparql to pig latin. In Proceedings of the International Workshop on Semantic Web Information Management, page 4. ACM, 2011.

47. Kim, H., Ravindra, P., and Anyanwu, K.: From sparql to mapreduce: The journey using a nested triplegroup algebra. Proc. VLDB Endow, 4(12):1426–1429, 2011.

48. Schätzle, A., Przyjaciel-Zablocki, M., Dorner, C., Hornung, T. D., and Lausen, G.: Cascading map-side joins over HBase for scalable join processing. RWTH, 2012.

49. Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th annual Symposium on Cloud Computing, page 5. ACM, 2013.

50. marathon: A container orchestration platform for mesos and dc/os", 2018.

51. Maschhoff, K., Vesse, R., and Maltby, J.: Porting the urika-gd graph analytic database to the xc30/40 platform. In Cray User Group Conference (CUG15), Chicago, IL, 2015.

52. Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I.: Graphx: A resilient distributed graph system on spark. In First International Workshop on Graph Data Management Experiences and Systems, page 2. ACM, 2013.

53. Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al.: Apache spark: a unified engine for big data processing. Communications of the ACM, 59(11):56–65, 2016.

54. Guo, Y., Pan, Z., and Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web, 3(2-3):158–182, 2005.

55. Morsey, M., Lehmann, J., Auer, S., and Ngomo, A.-C. N.: Dbpedia sparql benchmark–performance assessment with real queries on real data. In International Semantic Web Conference, pages 454–469. Springer, 2011.

56. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., and Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In Proceedings of the 17th international conference on World Wide Web, pages 595–604. ACM, 2008.

57. Husain, M. F.: Data intensive query processing for Semantic Web data using Hadoop and MapReduce. The University of Texas at Dallas, 2011.

58. mira — argonne leadership computing facility".

59. cooley — argonne leadership computing facility".

60. Gropp, W., Lusk, E., Doss, N., and Skjellum, A.: A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing, 22(6):789–828, 1996.

61. Momjian, B.: PostgreSQL: introduction and concepts, volume 192. Addison-Wesley New York, 2001.

62. Suchanek, F. M., Kasneci, G., and Weikum, G.: Yago: a core of semantic knowledge. In Proceedings of the 16th international conference on World Wide Web, pages 697–706. ACM, 2007.

63. Apweiler, R., Bairoch, A., Wu, C. H., Barker, W. C., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., et al.: Uniprot: the universal protein knowledgebase. Nucleic acids research, 32(suppl_1):D115–D119, 2004.

64. Beckett, D.: Rdf 1.1 n-triples. W3C recommendation, 2014.

65. Wilkinson, K., Sayers, C., Kuno, H., and Reynolds, D.: Efficient rdf storage and retrieval in jena2. In Proceedings of the First International Conference on Semantic Web and Databases, pages 120–139. Citeseer, 2003.

66. Neumann, T. and Weikum, G.: Scalable join processing on very large rdf graphs. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pages 627–640. ACM, 2009.

VITA

MICHAEL LEWIS

mlewis8@lewisu.edu

EDUCATION

DOCTORATE OF PHILOSOPHY IN COMPUTER SCIENCE December 2018

University of Illinois at Chicago (UIC), Computer Science Department and Electronic Visualization Laboratory (EVL) Chicago, IL

My thesis is titled "A Distributed Graph Approach for Retrieving Linked RDF Data Using Supercomputing Systems" . This work uses MPI and C++ on a 1000-node+ cluster supercomputer at Argonne National Laboratory to create a unique distributed framework that can transform a large dataset of subject, predicate, and referent statements into a traversable directed graph composed of connected nodes that store partial queries. The framework implements, join constructions and query retrievals on a graph structure in order to retrieve user-specified queries. From my thesis proposal, I received a Microsoft Azure Research Award (July 2014 – July 2015). Through my research, I have been able to utilize one of the most powerful super-computers in the world at Argonne National Laboratory (June 2015 to present). My research journey has also allowed me to travel the globe as an Open Science Data Cloud Fellow (2013-2014) to collaborate on big data with researchers in Europe and Brazil –See EXPERIENCE below. I received a Diversifying Higher Education Faculty in Illinois (DFI) fellowship from August 2011 – August 2015, which is a highly competitive award given by the State of Illinois geared for underrepresented students. The DFI award aims to foster and prepare award recipients to work in government-supported educational and research institutions in the State of Illinois. I will complete my PhD at the end of the summer 2017.

MASTER OF SCIENCE IN COMPUTER SCIENCE July 2002

UIC, Computer Science Department and EVL Chicago, IL

Thesis developed a C++ library to support collaborative menu interfaces within the CAVE™ virtual reality environment, which was developed and commercialized by EVL.

BACHELOR OF SCIENCE IN ENGINEERING December 1993

Department of Interdisciplinary Engineering, Purdue University West Lafayette, IN

PUBLICATIONS

Lewis, Michael J., et al. "A distributed graph approach for pre-processing linked RDF data using supercomputers." *Proceedings of The International Workshop on Semantic Big Data*. ACM, 2017.

PROFESSIONAL EXPERIENCE

LEWIS UNIVERSITY, ASSISTANT PROFESSOR May 2017 - Present

I became an assistant professor at Lewis University, August 21. Since teaching at Lewis, I have taught the courses: Machine Learning, Applied Programming Languages, Data Visualization, Capstone Software Systems, Data Analysis and Data Mining. The Capstone Software Systems class deals with working with community organizations in tandem with students divided into teams on a semester project.

LOYOLA UNIVERSITY CHICAGO, TEACHING INSTRUCTOR Jan 2016 - May 2017

Open Source Computing: This course, I taught in the fall semester 2016 and the spring semester 2017. The class composed of teams would endeavor in a semester project where each team in the class would create or improve on previous software used from an open source repository. Newly created software would have to be defensibly useable and/or expandable within the open-source community. The class also added discussions and writing assignments over cyber-space issues relating to privacy, regulation, security and the interpretation of freedom.

Introduction to Computing: This course was taught in the Spring semester of 2016. This course was the general course open to all students at Loyola University Chicago for the spring semester 2016. I instructed my students in Python programming, providing them with lectures, homework assignments and programming projects.

RESEARCH AID (INTERNSHIP), ARGONNE NATIONAL LABORATORY June 2015 - August 2016

Argonne National Laboratory is a multidisciplinary science and engineering research center, employing hundreds of Post-Doctoral Scholars, graduate and undergraduate students. It is host to one of the fastest super-computers in the world. As an intern (June 2015- August 2015) I developed python scripts, and wrote and expanded Fortran and C scientific simulation, MPI code to evaluate the I/O performance of the Mira supercomputer containing 65,536 cores. My research work provided Argonne performance scientist a better understanding of how to use application dependent algorithms and utilize MPI group IO functions to improve the read times of large scale job simulations. As a research assistant (August 2015 - August 2016), I worked with Argonne staff and a Loyola tenured professor in coordinating my thesis work with aspects of file I/O and node to node communication within a supercomputing environment.

ASSISTANTSHIPS AND RESEARCH WORK, UIC July 2009- May 2015

Graduate Assistant, Graduate College – Office of Recruitment and Diversity Aug 2012- May 2015

UIC is recognized as having one of the top Graduate programs in the country. As a Graduate Assistant, I assist in recruiting underrepresented, qualified UIC graduate school prospects from around the country. I help facilitate and organize tours, perform demos in the CAVE2™, a virtual-reality environment recently developed and commercialized by EVL. I also developed recruitment software that is actively used in recruitment outings; "E-Recruiter" is tablet-based software developed with Eclipse, Android Developer Tools. The software collects prospective student information, packages it in a spreadsheet data format, and provides a user interface to view the prospective student's info and to email follow-up letters on those selected students within the spreadsheet viewing interface.

Open Science Data Cloud (OSDC) Summer Fellowship June – August 2014

<http://pire.opensciencedatacloud.org/research/fellows2014/#michael-lewis>. As a part of my summer fellowship, I attended a Big Data training workshop in Amsterdam, Netherlands and then worked at the Laboratory of Computer Networks and Architecture (LARC) at University of Sao Paulo in Brazil for 6 weeks on collaborative research. My research used network technologies to optimize big data transfers. I also gave a presentation to Brazilian computer science students regarding my PhD research.

Figure 30: Resume

Open Science Data Cloud (OSDC) Summer Fellowship — June – August 2013

<http://pire.opensciencedatacloud.org/osdc-pire-year-2-researchers-2013>. I attended a Big Data training in Scotland, Edinburgh (July 7-14) to learn to configure, allocate and program computing nodes within a Cloud environment for the purpose of exploring very large datasets. I used the Open Stack cloud engine and Hadoop and R as tools to explore datasets. I then worked with researchers at the University of Sao Paulo in Brazil to use the Open Science Data Cloud Clusters on a Cloud framework to accumulate and analyze data from various weather stations in California and Brazil.

Future Grid Project with Indiana University — April 2011 – August 2011

Future Grid is a cloud computing test-bed developed by a consortium of Universities including Indiana University, Purdue University, and University of Chicago. I was responsible for having the University of Illinois at Chicago (UIC) be a contributor to Future Grid, specifically by developing a REST API to connect to and query a Cloud-based image deployed on Future Grid. My (UIC) team was composed of myself and one other student. The software tools that were used to create a REST image were: Python, CherryPy, OpenSSl, twill (for unit testing) and MongoDb for the database.

Teaching Assistant (Computer Graphics) — August 2010 – December 2010

Responsible for creating and lecturing on computer-graphics-based projects for students and grading laboratory work. Projects assigned included the following topics: perspective projection, polygon filling, ray tracing, and Phong shading.

Research Assistant (Lead Developer, Education Game Project) — May 2009 – September 2010

Recruited to develop a professional grade education software game to teach students on how to survive disasters. This game, developed with the Unity Game engine, features a camera-follow/first-person boy character. This character navigates around a Virtual house while receiving direction via menus on what to find and do in case of an earthquake. The language used in the project was Unity Script, similar to Java script. The features and modules that I created included developing an algorithm to simulate the earthquake movements for all the elements in the house, implementing the camera following movements and the occlusion module, and creating the menu interfaces. This game was funded by and distributed by the Illinois Emergency Management Agency and tested in a select number of Illinois schools.
http://public.iema.state.il.us/webdocs/earthquakegame/Welcome.html

SCIENTIFIC VISUALIZATION SPECIALIST — April 2005 – April 2007

NetASPX Inc. Army High Performance Computing Research Center — Minneapolis, MN

NetASPX is a database company, maintaining data and large-scale inventory for top tier corporations. The Army High Performance Computing Research Center (AHPCRC) was a division of NetASPX contracted by the Department of Defense for army research of vehicles and armor. I used C and C++ and MPI on supercomputers (CRAY and Linux clusters) to code a visualization application that retrieved Computational Fluid Dynamics (CFD) datasets pertaining to bullet penetrations on armored vehicles and vests. While working there I also received secret clearing status.

VIRTUAL REALITY SOFTWARE DEVELOPER — August 2001 – February 2005

Figure 31: Resume

Fuel Tech Inc. Virtual Vantage Software Division — Batavia (now Warrenville), IL

Fuel Tech is a company that provides a broad array of technologically advanced solutions to meet the pollution control and efficiency improvements for boiler, coal-burning companies. Acuitiv (Virtual Vantage), the former software division of Fuel Tech, used VR (Virtual Reality), networking and interactive computer graphics to develop a highly interactive and immersive Computational Fluid Dynamics (CFD) application designed to help companies understand problematic flows. As a VR software developer, I used C++ to develop the client server visualization software (Acuitiv). This software would read in CFD datasets to be navigated and viewed on a desktop or in a virtual environment. I coded modules to read different structured datasets, and to construct iso-surfaces, streamlines and contour planes from the extracted data. The CFD application developed by our software team received the 2004 product of the year award by Desktop Engineering. http://www.deskeng.com/articles/aaaats.htm

Software Inventions

E-RECRUITER — Developed: October 2013 – present

Office of Technology Management (OTM), UIC, Technology No. DH156 — Chicago, IL

University of Illinois at Chicago, is licensing the technology "E-Recruiter". This technology is based on Android-based tablet software, also named E-Recruiter, that I solely developed. The application allows prospective students to enter their academic information. The software allows the recruiter to list and query student information and provides an interface to automatically generate follow-up emails to a selected student prospect.

Hobbies

On my downtime, I enjoy playing the piano, cooking vegan foods and going to sessions of hot yoga!

Figure 32: Resume