BINARY:
A FRAMEWORK FOR BIG DATA INTEGRATION FOR AD-HOC QUERYING

by

Azadeh Eftekhari

A thesis submitted to the School of Computing
in conformity with the requirements for
the degree of Masters of Science

Queen’s University
Kingston, Ontario, Canada
(April 2016)

Copyright © Azadeh Eftekhari, 2016
Abstract

Traditional relational database systems are not practical for big data workloads that require scalable architectures for efficient data storage, manipulation, and analysis. Apache Hadoop, one of these big data frameworks, provides distributed storage and processing as well as a central repository for different types of data from different sources. Data integration from various sources is often required before performing analytics. Apache Hive on Hadoop is widely used for this purpose, as well as for data summarization and analysis. It has features such as a SQL-like query language, a Metastore to hold metadata and file formats to support access to various frameworks on Hadoop and beyond. For comprehensive analysis and decision-making, however, a hybrid system is required to integrate Hadoop with traditional relational database management systems in order to access the valuable data stored in relational databases. Current hybrid systems are either expensive proprietary products or require a system to be developed by the user, which requires programming knowledge. In addition these approaches are not sufficiently flexible to be applied to other frameworks.

In this thesis, we propose a framework called BINARY (A framework for Big data INtegration for Ad-hoc queRYing). BINARY is a hybrid Software as a Service that provides a web interface supported by a back-end infrastructure for ad-hoc querying, accessing, visualizing and joining data from different data sources, including Relational Database Management Systems and Apache Hive. Our framework uses scalable Hive and HDFS big data storage systems and supports different data sources via back-end resource adapters. There is also a front-end web interface that enables the use of HiveQL to query the data sources.

The framework is extendable and allows adding other storage engines (e.g. HBase) and analytics engines (e.g. R) as needed. We used REST software architecture to enable loose connections between the engines and the User Interface programs to facilitate independent updates without affecting the data infrastructure. Our approach is validated with a proof-of-concept prototype implemented on the OpenStack cloud system.
Acknowledgements

First and foremost I would like to sincerely thank my supervisors, Prof. Patrick Martin, and Dr. Farhana Zulkernine. I am grateful to you for your guidance and knowledge, and most of all for your valuable feedback, support and encouragement throughout my research.

I would like to thank Wendy Powley for offering her continuous support and helping me during the course of my research.

I am very grateful to Shadi Khalifa and Yehia Elshater for friendly technical discussions and constructive criticism during this work.

I would like to thank the School of Computing and the School of Graduate Studies at Queen's University for providing me with the opportunity to pursue my master's degree in computing. I am grateful to IBM Canada Ltd. and NSERC for their financial support to the Database Systems Lab at Queen's University.

Lastly, but most importantly, I would like to sincerely thank my parents and my dear husband, Alireza Safaee, for their continuous support and encouragement.
# Table of Contents

Abstract .......................................................................................................................... ii
Acknowledgements .......................................................................................................... iii
List of Figures ..................................................................................................................... vi
List of Tables ..................................................................................................................... vii
List of Abbreviations ....................................................................................................... viii

Chapter 1 INTRODUCTION ................................................................................................. 1
  1.1 Motivation .................................................................................................................... 3
  1.2 Research Statement ..................................................................................................... 5
  1.3 Contributions .............................................................................................................. 5
  1.4 Thesis Organization ..................................................................................................... 6

Chapter 2 BACKGROUND ................................................................................................... 7
  2.1 Hadoop ....................................................................................................................... 8
    2.1.1 Apache Hadoop Architecture .................................................................................. 8
    2.1.2 Yet Another Resource Negotiator (YARN) ................................................................ 9
  2.2 SQL-on-Hadoop ......................................................................................................... 11
    2.2.1 Apache Hive ......................................................................................................... 11
      2.2.1.1 Apache Hive Architecture ............................................................................... 12
      2.2.1.2 HCatalog ....................................................................................................... 13
    2.2.2 Apache Drill ......................................................................................................... 14
  2.3 Apache Sqoop ........................................................................................................... 15
  2.4 RESTful Web Service ............................................................................................... 15
  2.5 Related Work ............................................................................................................ 17
    2.5.1 Parallel Databases ............................................................................................... 17
    2.5.2 Parallel Processing Frameworks ............................................................................ 20
    2.5.3 SQL-on-Hadoop in Commercial Analytical Platforms ........................................ 23
      2.5.3.1 Hortonworks Data Platform (HDP) ................................................................. 23
      2.5.3.2 Cloudera Distribution including Apache Hadoop (CDH) ............................. 24
      2.5.3.3 MapR ............................................................................................................. 24
  2.6 Summary .................................................................................................................... 24

Chapter 3 CONCEPTUAL ARCHITECTURE ......................................................................... 26
  3.1 Architecture ............................................................................................................... 26
    3.1.1 User interface layer ............................................................................................. 28
List of Figures

Figure 1.1: Big Data integration scenario. ................................................................. 2
Figure 2.1: Hadoop architecture............................................................................ 9
Figure 2.2: Apache YARN ecosystem.................................................................. 10
Figure 2.3: Architecture of Apache Hive on YARN............................................. 13
Figure 2.4: HCatalog architecture....................................................................... 14
Figure 3.1: Binary – our proposed ad-hoc query architecture. . ......................... 27
Figure 3.2: Hive adapter and its relevant RESTful API codes called with a single URL ...................................................................................................... 30
Figure 3.3: Details of resource adapters connecting RESTful APIs and data resources. .................................................................................................. 30
Figure 4.1: OpenStack- Instance configuration ..................................................... 34
Figure 4.2: Prototype architecture....................................................................... 35
Figure 4.3: Initial screen ....................................................................................... 36
Figure 4.4: HiveQL as a uniform language in the resource adapters...................... 38
Figure 4.5: Web interface workflow..................................................................... 41
Figure 4.6: Workflow between Hive and MySQL using two Sqoop versions for reliability reasons................................................................. 43
Figure 5.1: Selecting a table in Hive................................................................. 47
Figure 5.2: Selecting and filtering attributes of a table in Hive. ......................... 48
Figure 5.3: Result of querying a table in Hive.................................................... 48
Figure 5.4: Querying a table in MySQL............................................................ 49
Figure 5.5: Result of querying a table in MySQL ............................................. 49
Figure 5.6: List tables in Hive metastore via HCatalog....................................... 50
Figure 5.7: Join between two tables in Hive..................................................... 51
Figure 5.8: Result of a join between two tables in Hive. .................................. 52
Figure 5.9: Join between two tables in MySQL.................................................. 52
Figure 5.10: Result of a join between two tables in MySQL............................. 53
Figure 5.11: Join between two tables in Hive and MySQL............................... 54
Figure 5.12: Result of a join between two tables in Hive and MySQL................ 54
Figure 5.13: Result of a join between two tables in Hive.................................. 55
Figure A.1: Jersey servlet configuration........................................................... 67
List of Tables

Table 2.1: Feature comparison chart........................................................................................................20
Table 2.2: SQL-on-Hadoop systems [36] ....................................................................................................23
Table 4.1: Components of the prototype system........................................................................................33
Table 4.2: Function calls included in the RESTful API.............................................................................39
List of Abbreviations

CDH  Cloudera Distribution including Apache Hadoop
CPCSSN  Canadian Primary Care Sentinel Surveillance Network
DBMS  Database Management System
DAG  Directed Acyclic Graph
EMR  Emergency Medical Records
ETL  Extract, Transform, Load
HATEOAS  Hypermedia As The Engine Of Application State
HDFS  Hadoop Distributed File System
HDP  Hortonworks Data Platform
JAX-RS  Java API for RESTful Web Services
JDBC  Java Database Connectivity
LLVM  Low Level Virtual Machine
MPP  Massively Parallel Processing
ODBC  Open Database Connectivity
OLTP  Online Transaction Processing
RDBMS  Relational Database Management System
RDDs  Resilient Distributed Datasets
REST  REpresentational State Transfer
RPC  Remote Procedure Call
SaaS  Software as a Service
SerDe  Serializer and Deserializer
SOAP  Simple Object Access Protocol
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAR</td>
<td>Web Application Archive</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>YARN</td>
<td>Yet Another Resource Negotiator</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

The concept of big data has evolved from the requirement of scalable architectures for efficient storage, manipulation, and analysis of very large and complex data sets that are characterized by the four Vs - volume, variety, velocity, and veracity [40]. These data are generated from nature thanks to inexpensive sensors available today which are connected to microprocessors, high-speed computers and networks, as well as from social networks and similar data resources. There is a need for the support of big data in government organizations, industries and academia to better manage their data, gain insights for business development and marketing, and provide better services to the customers.

While traditionally many organizations have relied on the large relational databases to store and process data using Database Management System (DBMS) inspired analysis tools, this is not a practical approach anymore for large-scale data analysis [15]. Big data analytics requires new methods for all levels of data processing, from raw data acquisition to in-depth data analysis, to enable enhanced decision making and discovering insights [24].

To handle and process large-scale data efficiently and economically the system should be scalable and capable of handling data streaming and storing on low-cost storage, and processing by commodity hardware. Apache Hadoop [6] is an open source framework designed to process huge amounts of data in batches on commodity hardware using the MapReduce paradigm, which has mapper and reducer phases, on the Hadoop Distributed File System (HDFS) [63]. Apache “Yet Another Resource Negotiator”, or YARN [74], is the next generation of Hadoop and is also designed to support non-MapReduce applications on HDFS.
Although enormous amounts of new data are being generated every day, many large enterprises such as banking systems, health care providers, governmental laboratories still have a considerable amount of data stored and maintained in traditional data storage systems, which now need to be integrated with the new big data storage systems for data analytics. Different analytics methods require data in different formats. Therefore, a data integration tool is required that can facilitate the transition of data from one tool to another when executing a data analytics workflow as shown in Figure 1.1. Such a workflow includes steps of extracting, restructuring, relocating or publishing data, in real-time or batch mode [33].

Figure 1.1: Big Data integration scenario.
1.1 Motivation

For decades companies and information professionals have relied on Relational Database Management Systems (RDBMS) such as MySQL [50], DB2 [44], or others, to store and retrieve their data. With the constantly increasing scale and rate of data generation and the availability of low-cost and reliable storage, many organizations are adopting big data technology paradigms. It is especially important that these paradigms should have higher flexibility for managing wider varieties of data types and providing access to the data stored in the traditional RDBMSs. Another issue with large and unstructured data is that the information from raw format cannot be effectively understood and processed unless it is processed by solutions relying on the extracted structured data [1].

An example of such a scenario is the healthcare industry. For a healthcare data analyst, an ideal software solution is the one that provides access to the current and past records of patients in a unified view with no concerns about how to access or merge the required data from various data sources. Such a solution should provide the possibility to browse and find the required data and only move the required data between different resources (not the whole data source). In addition, it should allow the analyst to merge the data and store the outcome.

The coexistence of many different types of legacy and big data storage systems [85] calls for data integration tools for efficient data analytics. The tools should support the integration of data from different distributed sources into a big data store with a low cost (based on open source tools), platform independence and high flexibility. Hortonworks [41], Cloudera [21], and MapR [48] are some of the examples of common data platforms for big data which require an installation of the system either on the user’s local machine or on a cloud server. The users need to know how to connect the data platforms above to a RDBMS and to be familiar with the specific framework language to perform operations such as queries.
SQL-Like languages on Hadoop allow users to conveniently specify what they want in a familiar language. Apache Drill [5], as one of the SQL-Like languages on Hadoop, enables interactive ad-hoc analysis of large-scale datasets with low latency. It has interoperability with several back-end datastores and metadata-stores. However, currently it has no connectivity to a commonly used RDBMS such as MySQL [50].

In summary, current storage and integration platforms have the following shortcomings with respect to big data:

- Inability to support hybrid databases consisting of both scalable big data storage and RDBMS types.
- Lack of a universal view of a variety of data formats to query and manipulate data from multiple data sources of different types.
- Lack of web access to facilitate ubiquitous access to data.
- Lack of a data integration tool that includes the above functions and provides support for both traditional RDBMSs and big data infrastructures including data conversion and flow control for multi-step data analytics.
- Lack of open source tools with no constraints on proprietary data formatting, coding and hardware requirements.

The motivation for this work, therefore, is driven by the need for users to be able to perform ad-hoc queries on big data with a flexible approach. It is desirable to have a consolidated view (e.g. through a web interface) for the end users to access and integrate data between RDBMS and HDFS with no programming requirement delivered in Software as a Service (SaaS). SaaS is a software distribution model that provides access to software remotely over the Internet as services with no requirement to install the software locally by the users on their own machines [26].
1.2 Research Statement

Given the shortcomings of current systems highlighted above, the goal of our research is to develop a conceptual framework to support ad-hoc query on big data [65] that is based on open source tools and that meets the following requirements:

1. Consolidates data from different distributed sources into a big data store.
2. Supports a variety of data sources.
3. Provides a user-friendly web interface to access and query data with no need for programming.

The web interface provides an alternative to the common method of using drag and drop symbols for integrating modules in a proprietary software environment such as Cloudera, Tableau [22] and Hortonworks Datameer [42].

1.3 Contributions

In this research, we present BINARY, a conceptual framework that allows users to view and integrate data from an RDBMS into HDFS using Apache Hive with no need to import the complete data set into Hive. BINARY is platform-independent and works with different storage types and gives more flexibility to the solution compared to existing solutions. In addition, BINARY is extendable, which allows adding more data sources as needed. Note that throughout this thesis (excluding section 2.1.1) we follow the definition of client as an application on the user’s local machine that relies on a remote server to execute certain operations.

We have implemented a prototype of BINARY in this research that:

i. Integrates data from different data sources into HDFS on the back-end using Apache Hive;
ii. Provides RESTful APIs for all the data frameworks used on Hadoop and RDBMS to facilitate independent updating of the front-end programs without affecting the back-end data infrastructure; and

iii. Provides a web interface to the end users to access and join data between RDBMS and HDFS with no requirement of programming knowledge or installation of any software.

1.4 Thesis Organization

The thesis is organized as follows. The background and related work for the research is discussed in Chapter 2. Chapter 3 describes the conceptual architecture of the proposed software solution. Implementation details are discussed in Chapter 4. Validation with several test case scenarios is presented in Chapter 5 and Chapter 6 concludes the thesis and provides recommendations for future work.
Chapter 2

BACKGROUND

Ad-hoc ("for the purpose" in Latin) analysis in business-intelligence is designed to get information upon demand. In other words, the business-intelligence user desires to extract insights from historical trends or by finding correlations between data stored in multiple data sources. Ad-hoc querying is often an essential operation in data warehouses and can be quite complex. It is performed in the intervals between periods of loading new data as a typical warehouse operation. Statistical models and analytical reports are usually the eventual outcome of prior ad-hoc analysis [68].

Most of the time the user cannot know the queries that are needed beforehand or the circumstances can change over time. As the size of data grows, increased physical resources are required, which brings challenges to ad-hoc analysis. The main requirements for ad-hoc analysis over terabyte scale data (or larger), including the legacy data, are guaranteed connectivity to various data sources and availability of the related tools to perform the query. Having an extendable system is important for using the available resources efficiently with the lowest operational cost [30]. This research discusses the challenges of accessing, analyzing and manipulating data with a focus on ad-hoc querying over different sources involving big data and providing a unified view for the users.

In this chapter, we present the necessary background for the research including the Hadoop framework [81], SQL-on-Hadoop [1], Apache Hive [71], Apache Sqoop [12],[72], and RESTful web services [30]. We also introduce Apache Drill [36] including SQL-on-Hadoop approaches that are often used on analytical platforms.
2.1 Hadoop

Apache Hadoop [81] is an open source framework for storing, analyzing and transforming huge amounts of data in a distributed manner across low-cost machines. Apache Hadoop has become a popular solution for integrating different data processing tools as it provides high scalability for computation, storage and IO bandwidth [24].

2.1.1 Apache Hadoop Architecture

The Hadoop architecture is presented in Figure 2.1. It has two major layers, a storage layer with Hadoop Distributed File System (HDFS), and a processing layer (MapReduce).

HDFS stores massive amounts of data in a distributed form with high availability and strong fault tolerance. It consists of two types of nodes, NameNodes and DataNodes. NameNodes maintain metadata for all the files and directories in the file system tree. The application data is stored in DataNodes [81].

MapReduce processes a large amount of data in a distributed way using the Map and the Reduce phases. JobTracker and TaskTracker [6] are two major components of the MapReduce layer. The JobTracker is a master node, which runs with the NameNode. It receives the MapReduce jobs, locates TaskTracker nodes to execute the jobs, delegates the jobs to the TaskTracker nodes and monitors the execution status of the jobs.

A TaskTracker is a slave node, which runs on each DataNode. It accepts the jobs from the JobTracker, executes them and communicates with the JobTracker to report progress.
2.1.2 Yet Another Resource Negotiator (YARN)

YARN [74] is an open source resource management layer for the Apache Hadoop computing platform. It has solved two drawbacks of the original Apache Hadoop, namely:

- Restriction of exclusive use of the MapReduce programming model,
- Lower performance of the tasks because the JobTracker system performs parallel tasks of scheduling, monitoring the jobs as well as monitoring the cluster resources.

YARN separates the resource management functionality from the programming model. It also introduces per-job components to perform many scheduling-related functions. The JobTracker's tasks are split between the Resource Manager handling the cluster's resources and the Application Master, a per application master tracking the status of a submitted job in the YARN cluster. Also, YARN uses the NodeManager instead of the TaskTracker of Hadoop to replace the fixed time slot mechanism [74].

The Apache YARN ecosystem, as shown in Figure 2.2, is a set of services capable of running on top of Apache YARN. Each of the services takes advantage of HDFS e.g. storing huge amounts
of data with efficient fault tolerance. This ecosystem supports multiple data processing engines i.e. batch, interactive and real-time [6].

<table>
<thead>
<tr>
<th>BATCH, INTERACTIVE and REAL-TIME DATA ACCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
</tr>
<tr>
<td>Apache Hive</td>
</tr>
</tbody>
</table>

YARN: Cluster Data Management

Hadoop Distributed File System (HDFS)

Figure 2.2: Apache YARN ecosystem

The main components of the YARN ecosystem are introduced below:

i. Apache Hive [71] is an open source data warehouse infrastructure to analyze a large amount of batch data. It stores structured data in HDFS as a schema-oriented database. Details about Hive are provided in section 2.2.1.

ii. Apache Pig [52] is an open source engine to analyze batch data in large scale with a scripting language referred to as Pig Latin. The compiler in Pig translates the sequence of jobs in Pig Latin to MapReduce jobs.

iii. Apache HBase [32] is an open source distributed non-relational database similar to Google’s BigTable [17] for storing large quantities of sparse data. It is based on Hadoop
and inherits the scalability and fault tolerant mechanism of HDFS. Facebook uses HBase for its messaging service system [35].

iv. Apache Spark [84] is a fast in-memory processing system. It is designed to process streaming and batch data with high fault tolerance using Resilient Distributed Datasets (RDDs) [84]. Spark is applied as run-time for SQL processing in Shark [83], Hive on Spark [38], and Spark SQL [14].

v. Apache Storm [58] is an open source distributed real-time system to process large scale of real-time data. It has high scalability and fault-tolerance and has been adopted for online machine learning, ETL, real-time analytics and more. It can be used with many programming languages.

2.2 SQL-on-Hadoop

Many of the enterprise data management tools use HDFS as a central repository for data from various data sources. These enterprises rely on SQL and their users are familiar and comfortable with it because SQL processors have removed the low-level programming requirements for developers that are difficult to maintain and reuse. As a result, SQL processing over Hadoop data, referred to as SQL-on-Hadoop, has gained significant attention both in academia and industry [1]. In the following subsections some SQL-on-Hadoop engines are introduced.

2.2.1 Apache Hive

A data warehouse is a central repository to gather data from various data sources [64]. Apache Hive is an open-source data warehouse infrastructure originally developed at Facebook [27] with a SQL-like declarative language, referred to as HiveQL. It is built on top of Hadoop YARN and inherits its advantages such as scalability, availability and fault tolerance. Hive allows
summarization, query, and analysis for a large amount of batch data. Hive queries are translated into map-reduce jobs after being parsed, compiled and optimized to produce a physical query execution plan in the form of a Directed Acyclic Graph (DAG) of MapReduce tasks. Hive also has a system catalog useful for exploring data and optimizing queries. Hive has a Serializer and Deserializer (SerDe) functionality to read and write table rows (with different file formats) from and to HDFS format [8].

2.2.1.1 Apache Hive Architecture

The Apache Hive architecture is presented in Figure 2.2. Hive [71] on the Hadoop YARN engine consists of the following components.

i. External Interfaces: Hive queries can be run from user interfaces (UI), e.g. command line, web UI or an application programming interface (API) such as Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC).

ii. Hive Thrift server: It provides client APIs that get installed on user’s machine, and support different languages and communicate with Hive services running on the server.

iii. Metastore: It keeps metadata about the stored tables in Hive, which are specified when the associated tables are created. A metastore is used every time HiveQL calls the table.

iv. Driver: The driver handles the life cycle of a HiveQL statement by creating a session for compilation, optimization and execution.

v. Compiler: It provides a plan for the HiveQL statement which is a DAG of MapReduce tasks.

vi. Execution Engine: The DAG provided by the compiler is submitted to the execution engine in a topological order.
2.2.1.2 HCatalog

HCatalog is a storage management layer built on top of Hive metastore that enables Apache Pig [52], Hive, and MapReduce users to share schemas by providing a relational view of the data stored on HDFS. HCatalog has a REpresentational State Transfer (REST) interface [59] (via WebHCat [80]) that gives the external systems access to Hive metadata. HCatalog supports reading and writing various file formats (e.g. RCFile, CSV, JSON, and SequenceFile formats) using Hive SerDe [37]. HCatalog architecture is shown in Figure 2.3.
For interactive SQL queries on Hive the latency of launching jobs per query is too long and sometimes impractical. For such reasons, we see a shift toward shared-nothing database architectures for SQL processing over Hadoop data [1]. A shared-nothing architecture is a distributed computing architecture with independent and self-sufficient nodes that reduces the conflict of using shared resources and thus has a better performance. Due to the co-existence of a long-running process and DataNodes on each node in the cluster, the SQL queries can be answered instantly. Apache Drill [5] is an example of these Massively Parallel Processing (MPP) architectures and is discussed in the next section.

2.2.2 Apache Drill

Apache Drill [5] is an open-source ‘interactive’ SQL query engine for Hadoop that provides fast interactive queries for large datasets. Working with schemaless data is the main advantage of Drill among other SQL-on-Hadoop counterparts. Drill can start the query execution process with no need for schema or type specification. Self-describing data (e.g. JSON, Parquet) can be handled using the SQL-like declarative processing, and data from files and HBase tables can be discovered at the time of reading them. Drill discovers the schema of a fixed chunk of data, generates a columnar representation in-memory as well as specific code for processing. This way Drill can accommodate data chunks with varying schemas [1]. Drill supports multiple back-
end data stores such as Hadoop, NoSQL and Cloud storage systems [82] and has a focus on complex nested datasets (like JSON). Currently, a shortcoming of Drill is the lack of a connector to RDBMSs. On the other hand, the extensible architecture of Drill allows for storage plugins for specific requirements with a RESTful API as the recommended architectural style for plugins [5]. This research addresses the requirement of connecting Drill to RDBMSs while taking advantage of the RESTful API.

2.3 Apache Sqoop

Hive uses Apache Sqoop to import and export data from and to RDBMSs. Sqoop supports bulk data transfer between Hadoop and external relational databases and Hadoop and NoSQL systems. Sqoop is independent of operating systems and its connector-based architecture supports connectivity to additional external systems. It uses MapReduce to import and export data. Sqoop’s import process relies on the selected database or table to describe the schema. The selected table is read through Sqoop row-by-row and the output is saved on HDFS in multiple files. Similarly, for exporting data from HDFS to a target database table, Sqoop reads the files, parses them into records and then inserts them as new records in the target table [12].

2.4 RESTful Web Service

As discussed in section 2.2.2, a RESTful API is the recommended interface of Apache Drill. A web service is a common method for integrating web-based applications over the Internet. It is a cost-effective and practical computational software solution for unifying applications and data distributed over operating systems and platforms [56]. It enables exchange of information between units of an internal business or partners, and/or exposing existing functions on the network by acting as a wrapper and making itself accessible via standard protocols.
A RESTful web service [49] is a web service that supports REST architectural principles. REST has gained widespread acceptance compared to SOAP (Simple Object Access Protocol) or Web Services Description Language (WSDL) due to characteristics such as simplicity, flexibility, and high scalability. It makes the integration of an application with other frameworks more straightforward. Big companies such as Google, Amazon, eBay, and Yahoo use the RESTful architecture in their offered web services. For each resource function designed with RESTful web services, there is a unique URI to provide access to that resource. The key features of RESTful architecture include [30]:

i. Addressability: Every resource in the designed service is accessible through URIs directly or the hyperlinks that generate interconnectivity between related resources.

ii. A uniform constrained interface: Limited well-known HTTP methods are used to manage the resources including GET, PUT, DELETE, and POST with no further configuration for HTTP client. The uniform interface brings familiarity, interoperability, and scalability beneficial for the users, developers, and system owners respectively.

iii. Representation-oriented: Resources represented by URIs can have different formats. Representations in applied formats are supported, e.g. HTML is used for browsers, JSON is applied for JavaScript and so forth.

iv. Stateless communication: Client session data is not saved on the server. The server handles only the state of exposed resources, which simplifies the application scalability. RESTful approaches are self-contained and self-descriptive.

v. Hypermedia As The Engine Of Application State (HATEOAS): Hyperlinks can be used to link between resources. Referrals to hyperlinks are also applied to manage the state transfer.
2.5 Related Work

The problem of accessing various data sources and providing users with a unified view has been addressed via data integration for the web [25],[18] (e.g. pay-as-you-go [47], entity matching and resolution [19], user feedback and crowd-sourcing [16]) and for data analytic environments (e.g. data warehouses) [33]. Data integration plays a key role in analytics over big data that aims toward extracting meaningful knowledge from large-scale repositories of different types of big data [78]. Query engines have been developed for large-scale databases and parallel processing frameworks to provide transparent access to data sources by hiding the type and distribution of the sources (e.g. RDBMS and non-relational databases). In this section, some data integration techniques in parallel databases and parallel processing frameworks for in-memory or disk-based data sources are discussed. Also, some of the SQL-on-Hadoop approaches used in analytical platforms are presented.

2.5.1 Parallel Databases

To enhance the overall performance, in a parallel database system some operations are parallelized such as getting queries, generating indexes etc. Parallel database systems can run on clusters of shared-nothing nodes [54]. Standard relational tables and SQL are supported in such systems and the fact of data residing on multiple machines is not visible to the end-user [54].

Depending on the application, either a column-based or a row-based data storage architecture is favored. A row-oriented architecture is more suitable for Online Transaction Processing (OLTP). On the other hand, the main requirement for data analytics is the frequent reading of relevant data, and thus, a read-optimized column-oriented architecture is preferred for such applications. However, these systems are not optimized for joining and writing data [85].

SAP HANA [28] is a commercial column-oriented in-memory relational database offered by SAP [60]. It is intended to integrate transactional and analytical workloads within the same
DBMS. SAP HANA is equipped with a combined column and row engine that allows storage of relational data in tables either in a column or a row-oriented layout. Conversion from one layout to another is possible in order to express queries with tables in both layouts. SAP HANA has an extensible architecture to allow additional engines to support a wider variety of data sources. It also has a connector framework to link with external language packages such as R [29] and it supports database connectors (e.g. JDBC or ODBC). The data from all engines is kept in the main memory in a compressed form using a variety of compression schemes as long as the memory space is sufficient. Otherwise, it is reloaded into main memory as needed. Descriptions of data structures and tables are kept in a repository called the metadata manager that can be local or global. The challenges associated with in-memory column-oriented distributions include the necessity for data partitioning, support for distributed transactions and managing the demanding process for merging updates into the read-optimized storage layout [60]. SAP HANA does not provide a web interface and users need to know programming language to handle their requests. It is an in-memory database and hence it has the challenges of processing data larger than the available physical memory. Also, SAP HANA supports a subset of standard SQL.

Vertica is a disk-based analytic database system based on the C-Store [69] project which is now offered by HP [43]. It is a MPP distributed system with a shared-nothing architecture. Vertica has a column-oriented storage that keeps data in a compressed form using different encoding schema (e.g. Auto, RLE, Block Dictionary). Vertica keeps its metadata in memory (and not in database tables) and transfers it to disk via its own mechanism. Vertica initially stores data as tables of columns and later as segments on the database nodes. Vertica has a bulk loader and provides full SQL for querying. The SQL queries can be submitted via a vsq1 command prompt or through standard JDBC, ODBC, or ADO.net drivers. Integration of diverse data is done using drivers as database connectors and data warehouse methods. Users are required to know a programming language to manage data in Vertica.
Brighthouse [66] (recently renamed to Infobright) is a commercialized column-oriented data warehouse for ad-hoc analytic queries. It is based on Rough Sets where data is grouped into Packs with 64k values of single columns. The information about Data Packs and their relationships is stored in the metadata layer as additional data instead of classical indexes which is referred to as the Knowledge Grid. Knowledge Nodes are elements of the Knowledge Grid that describe data packs. The main feature of Knowledge Nodes is to determine unwanted data packs during query resolving. Using the Knowledge Grid, a Brighthouse query determines relevant, irrelevant and suspected packs. There are proprietary data storage and query optimization layers which allow integration of a Brighthouse database with MySQL. The use of data instead of indices in the Knowledge Grid for ad-hoc querying scenarios raises questions about the capability of Brighthouse to tune the physical database model. Brighthouse supports only structured data. Similar to the previous systems, the users need to know programming language to handle data.

Table 2.1 presents a feature comparison chart for the parallel databases discussed in this chapter, partially augmented and adapted from [70].

In general, parallel DBMSs follow rigid schemas or the relational archetypes of rows and columns. This makes it difficult to accommodate growing data (beyond certain limits) and hence consolidating various data resources. Also, they require high-end server hardware. In the case of a single node failure during a long query, the whole query must be repeated [54]. Parallel processing frameworks (e.g. Hadoop MapReduce, Spark) address these issues.
Table 2.1: Feature comparison chart

<table>
<thead>
<tr>
<th>Name</th>
<th>SAP HANA</th>
<th>Vertica</th>
<th>Brighthouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>In-memory, column based data store.</td>
<td>Column-oriented Analytics DBMS</td>
<td>Column-oriented RDBMS (with a focus in machine-generated data)</td>
</tr>
<tr>
<td>Database model</td>
<td>RDBMS</td>
<td>RDBMS</td>
<td>RDBMS</td>
</tr>
<tr>
<td>License</td>
<td>Commercial</td>
<td>Commercial</td>
<td>Commercial</td>
</tr>
<tr>
<td>Server operating systems</td>
<td>Appliance or cloud-service</td>
<td>Linux</td>
<td>Windows Server, Linux</td>
</tr>
<tr>
<td>Data scheme</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL</td>
<td>Support major part of Standard SQL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>APIs and other access methods</td>
<td>JDBC, ODBC</td>
<td>ADO.NET, JDBC, ODBC</td>
<td>Uses the MySQL drivers for ODBC, JDBC, .NET, etc.</td>
</tr>
<tr>
<td>Transaction concepts</td>
<td>ACID</td>
<td>ACID</td>
<td>-</td>
</tr>
<tr>
<td>In-memory capabilities</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Support MapReduce</td>
<td>No</td>
<td>Yes (via connector)</td>
<td>Yes (via connector)</td>
</tr>
</tbody>
</table>

2.5.2 Parallel Processing Frameworks

Apache Hadoop and Spark, the parallel processing frameworks introduced in section 2.1 and section 2.1.2 respectively, support various SQL-like engines. We describe some of these engines below.

Shark [83] is an open source data analysis engine running on Spark. Shark provides an interactive SQL-like language on Hadoop by using in-memory caching, columnar storage and compression schemes such as dictionary encoding and run-length encoding. It uses Hive’s language and catalogue to store its metadata and query external data. However, Shark faces challenges to run over Spark since Hive optimizer is not developed for Spark.

Spark SQL [14] is an open source SQL-like engine on Spark that has the advantages of
Shark such as the columnar storage and in-memory caching without its limitations such as dependence on the Hive metastore and the difficulty of running a query in Spark. Spark has a declarative API that allows integration of relational operators (e.g. projection) with the procedural Spark code (e.g. Scala, Java). This integration enables users to simplify expressing their query logic with more powerful tools than just SQL. Spark SQL defines a DataFrame, which is a distributed set of rows with an identical schema, similar to a table in RDBMS for its operations. Resources such as MySQL can exchange data with Spark either via JDBC or the declarative API. Spark has the common downside of in-memory systems i.e., the challenges in handling data larger than available physical memory and the susceptibility to power interruptions.

In addition to Hive and Drill (introduced in section 2.2), there are other SQL-on-Hadoop systems. Some of the popular ones that work with a shared-nothing architecture include Hadapt [34], Cloudera Impala [77] and IBM Big SQL [61] are discussed below.

Hadapt is a commercial outcome of the HadoopDB research project [72], one of the earliest SQL-on-Hadoop solutions. Hadapt and HadoopDB use DBMS-oriented storage, e.g. column-store data layouts instead of file-oriented HDFS storage formats. Hadapt applies two distinct run-time query engines: an interactive MPP run-time for shorter queries and a MapReduce-based run-time for longer queries where fault tolerance is needed. However, Hadapt is a proprietary product and requires a license.

Cloudera Impala [77] is an open-source MPP SQL query engine. Impala is capable of reading data with the bandwidth of all available disks. The main feature of Impala is to generate code at runtime to accelerate the frequently executed code paths and this is done using Low-Level Virtual Machine (LLVM) [77], a compiler infrastructure with reusable libraries and well-defined interfaces. The performance gained by this approach is, however, limited to the number of queries it can fit into the physical memory of the machine [31]. Impala includes a subset of SQL
functions and does not support RDBMS.

IBM Big SQL [61] applies IBM's latest relational database system to run the standard SQL queries on HDFS data while supporting all the Hadoop file types. Hive catalog, table definitions and metastore are used in Big SQL 3.0. Query rewrite transformations [62],[86] are quite intricate to support complex queries. A query execution plan is chosen through sophisticated statistical methods and a cost-based optimizer. Similar to Hadapt, IBM Big SQL is a proprietary system and requires a license.

A summary of SQL-on-Hadoop systems including Hive, Drill, Hadapt, Cloudera Impala and IBM Big SQL are represented in Table 2.2, even though every Hadoop distribution includes Hive. Among these systems, only Hive and Drill are non-proprietary and Hive, Drill and Impala are open source. Drill is the only system in which having a schema is optional. In contrary to Hadapt, Hive has compatibility with columnar storage, which is already implemented in other three systems.
Table 2.2: SQL-on-Hadoop systems [36]

<table>
<thead>
<tr>
<th>Owner</th>
<th>Apache Drill</th>
<th>Apache Hive</th>
<th>Impala</th>
<th>IBM Big SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Latency</td>
<td>Community</td>
<td>Community</td>
<td>Hadoop</td>
<td>Cloudera</td>
</tr>
<tr>
<td>Operational Mode</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data Shape</td>
<td>Nested, Tabular</td>
<td>Nested, Tabular</td>
<td>Tabular</td>
<td>Tabular</td>
</tr>
<tr>
<td>Data Sources</td>
<td>Extensible, including HDFS, HBase, Cloudera, MongoDB, etc.</td>
<td>HDFS, HBase,</td>
<td>HDFS, RDBMS</td>
<td>HDFS, HBase, RDBMS, etc.</td>
</tr>
<tr>
<td>Hadoop Dependent</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Schema</td>
<td>Optional Apache 2.0</td>
<td>Required Apache 2.0</td>
<td>Required Commercial</td>
<td>Required Commercial</td>
</tr>
<tr>
<td>License</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source Code</td>
<td>Open Extensible, including SQL 2003, MongQL, DSL, etc.</td>
<td>Open HiveQL</td>
<td>Closed SQL subset</td>
<td>Closed SQL 2003/2011</td>
</tr>
<tr>
<td>Query Language</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Columnar Storage</td>
<td>Yes</td>
<td>Possible</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

2.5.3 SQL-on-Hadoop in Commercial Analytical Platforms

In this section, we present the SQL-on-Hadoop systems for some of the commercial big data platforms such as Hortonworks [41], Cloudera [21] and MapR [48].

2.5.3.1 Hortonworks Data Platform (HDP)

HDP [41] is an open source data platform powered by Apache Hadoop and established by a company called Hortonworks. HDP’s environment integrates Hadoop with the other Apache components (e.g. Hive, Pig, HBase). YARN and HDFS are the core components of HDP and SQL queries run on HDP through Apache Hive. To access HDP, users need to install it on the cloud or on a virtual machine (VMware [76], VirtualBox [75]) and pass through the complexity of the initial set up and configuration. An RDBMS can be later configured to work with HDP.
Users need to know the specific language of each individual engine in order to work with them in the platform. Also, there is no web interface.

2.5.3.2 Cloudera Distribution including Apache Hadoop (CDH)

CDH [21] is an integrated platform for managing large-scale data using the Hadoop ecosystem offered by Cloudera Inc. Cloudera uses proprietary software for some of its components (e.g. Cloudera Impala). CDH offers Cloudera Impala for interactive SQL queries and Hive (HiveQL) for batch processing. Users need to install CDH on the cloud or on a virtual machine. They also need to know a specific language for each engine.

2.5.3.3 MapR

MapR [48] is another data platform with the industry-standard Hadoop distribution. It has many components of the Hadoop ecosystem (e.g. MapReduce, Spark, etc.). MapR replaces HDFS by MapRFS, which has its own file system. Hive, Impala, and Shark are applied as SQL processors in MapR. Apache Drill is deployed in the enterprise version of MapR. The platform can run on a virtual machine either as a free desktop application or on the cloud and in both cases users should do the configuration and know the required programming language of the engines in the platform.

2.6 Summary

This chapter describes the Apache Hadoop MapReduce framework and how it overcomes the limitations of traditional database technologies (e.g. scalability and failover) to support big data. Data integration concepts for big data are described from multiple perspectives: storage, query and parallel computing frameworks. The need for ad-hoc querying on big data sources is explained. The most common variations of open-source and commercial SQL-on-Hadoop processors, their advantages and limitations are studied. None of the popular data platforms
satisfy all our requirements of supporting both traditional RDBMS and big data, enabling query on a variety of data resources, facilitating ubiquitous access through a web interface, and using open source tools with no proprietary data formatting, coding or hardware.

A successful solution should provide universal accessibility to various data resources (RDBMS and HDFS) both in interactive and batch modes, and be independent of the hardware and operating system. The next chapter presents our proposed framework that addresses the above concerns.
Chapter 3

CONCEPTUAL ARCHITECTURE

We propose the BINARY framework to satisfy the requirements for ad-hoc querying on big data that are given earlier in the thesis. The conceptual architecture of our framework is explained in this chapter. The prototype implementation with illustrations using some use case scenarios are presented in the next chapter.

3.1 Architecture

We propose a multi-layer SaaS architecture for the ad-hoc query service as shown in Figure 3.1. It is composed of a User interface layer, an Application layer and a Resource layer.

1. The User interface layer provides a connection between the end users and the Application layer.
2. The Application layer provides a connection between the User interface and the Resource layer. This layer includes a set of RESTful APIs, which we refer to as the front-end.
3. The Resource layer contains Extract, Transform, and Load (ETL) resources, analytics and storage engines as well as RDBMS. These resources are also referred to as the back-end resources. These layers are discussed in detail in this chapter.
Figure 3.1: Binary – our proposed ad-hoc query architecture.

The BINARY framework components include:

i. Big data distributed storage: to store huge amounts of data in a distributed form with a high availability and strong fault tolerance working with processors handling a large amount of data in a distributed way,

ii. Back-end resource manager: to work seamlessly with the distributed storage and allow different programming models to be realized on the system,

iii. Scalable big data storage: to use distributed storage paradigm as a central repository for data from various data sources, capable of importing RDBMS data and having a meta-store for metadata to support analysis of large amounts of data (via SQL processing).

iv. ETL resource: to transfer bulk data efficiently from the RDBMS to distributed storage and vice versa.
v. Engines and data sources:
   - Storage engines: to access various data sources
   - Analytics workflow engines: to use storage engines and RDBMS to derive models and patterns from available data
   - RDBMS

vi. Adapters: to provide access to the Resource layer for data transfer, analytic workflow and storage on the front end,

vii. RESTful API: to provide a common user interface between the resource adapters and the User interface layer,

viii. User interface: to connect and serve end users request to the Application and Resource layers.

In the BINARY framework, we use Big data distributed storage and scalable big data storage to satisfy the requirement of scalability. BINARY also uses ETL tools for bulk data transfer between RDBMSs and distributed storage. We developed resource adapters to support different data sources to fulfill the variety requirement of ad-hoc querying. These include an adapter for the scalable big data storage in order to integrate data from different data sources into the distributed storage. It also enables data access and querying using a SQL-like language, which meets the ease-of-use requirement. The following subsections have more details for each layer in the BINARY architecture.

3.1.1 User interface layer

The custom user interface connects the users to the Application and Resource layers and serves their requests via SaaS. This layer is dependent on the intention of the application (e.g. accessing available engines and data resources and visualization). The user interface for the prototype is described in the next chapter.
3.1.2 Application Layer

The Application layer is located between the User interface and the Resource layers. It has two components as depicted in Figure 3.1:

- RESTful APIs, to connect the User interface layer and the resource adapters,
- Resource Adapters, to access to the back-end resources.

3.1.2.1 RESTful API

We developed RESTful APIs (see section 2.4) as interfaces between the user interface layer and resource adapters. The REST software architecture enables loose connections between the engines and user interface programs in order to facilitate their independent updates without affecting the back-end data infrastructure. The key features of a RESTful API include addressability, stateless communication and Hypermedia As The Engine Of Application State (HATEOAS). The User interface layer dispatches a URL such as

```
http://localhost:8080/HadoopREST/rest/hive/joinTable?ip=130.15.6.122&table1=tableName&table2=tableName&attribute1=attributeNames&condition1=table1attribute1=table2attribute3 &userName=root&passWd=myPassWd
```

through the RESTful APIs which directs the request to the appropriate resource adapter (see section 3.1.2.2 for details). The adapter invokes the appropriate function to execute the job.

In the example, by providing the username, password, IP address, and the selected table names in the URL, the user can get the content of the joined tables and view it on a web page (see section 4.2.2 for more detail and security consideration). Thus RESTful APIs can facilitate accessibility to a great extent. For instance, both the code fragments in Figure 3.2, can be accessed with a single line of URL.
3.1.2.2 Resource Adapters

In BINARY, a resource adapter manages the tasks of transferring and routing data from/to each back-end resource in the Resource layer (see section 0). Each adapter used in the application layer has a dedicated RESTful API as shown in Figure 3.3.

Figure 3.3: Details of resource adapters connecting RESTful APIs and data resources.

```
30
3.1.3 Resources Layer

We place the engines and data sources in the resources layer. They support the deployment of physical resources (local and remote machines, servers, etc.). The engines work with a resource manager and a file system manager to allocate, manage and manipulate data and other hardware and software resources. The ETL resource supports the bulk data transfer between RDBMSs and big data storages.

3.2 Summary

This chapter introduces the conceptual architecture of the proposed BINARY framework with the detailed descriptions of its components. The architecture includes three main layers: the User Interface layer which interacts with the end user, the Application layer which includes the RESTful API and the resource adapters, and the Resource layer containing the data resources. The user interface serves the end user by accessing the data resources through the RESTful APIs and the resource adapters. BINARY offers the possibility to work with both RDBMSs and scalable big data storage systems to integrate data and perform ad-hoc querying.
Chapter 4

PROTOTYPE IMPLEMENTATION

In this chapter we describe our prototype implementation to demonstrate that our BINARY framework can be used to satisfy the requirements for ad-hoc querying on big data that we identified in Chapter 1. The User Interface layer of the conceptual architecture (Figure 3.1) is web-based for convenient remote access. Through this interface the end users can perform their queries and write data in both RDBMSs and Hive, visualize the data in tabular forms, and execute ‘join’ operations of tables from different types of data sources.

This chapter explains the hardware and software setup used to implement our framework. Next, we describe the implementation and functionality of each of the components of the layout in terms of the specific software used in the prototype.

4.1 Setup

The prototype system is realized on a private cloud using the available OpenStack cloud platform [53] in our database lab at Queen’s School of Computing. OpenStack is open-source software to create and manage cloud environments with large-scale computational, storage, and network resources for cloud infrastructures. The software solution is deployed on a virtual machine with 2 virtual x86-64 CPUs, 4GB of RAM, and 40GB disk storage as shown in Figure 4.1 that runs Ubuntu. PuTTY [57] is used to access our Linux host remotely from Windows 8 platform. The details of the software components applied in this prototype are listed in Table 4.1.
Table 4.1: Components of the prototype system

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenStack</td>
<td>2014.1</td>
<td>[53]</td>
</tr>
<tr>
<td>Hadoop</td>
<td>2.2.0</td>
<td>[7]</td>
</tr>
<tr>
<td>Hive</td>
<td>0.12.0</td>
<td>[9]</td>
</tr>
<tr>
<td>HiveServer2</td>
<td>0.12.0</td>
<td>[39]</td>
</tr>
<tr>
<td>Sqoop</td>
<td>1.4.4, 1.99.3</td>
<td>[67]</td>
</tr>
<tr>
<td>Ubuntu Linux</td>
<td>12.04</td>
<td>[73]</td>
</tr>
<tr>
<td>MySQL</td>
<td>5.5</td>
<td>[50]</td>
</tr>
<tr>
<td>Connector/J, JDBC driver (mysql-connector-java)</td>
<td>5.1.31</td>
<td>[51]</td>
</tr>
<tr>
<td>JDK</td>
<td>1.7</td>
<td>[45]</td>
</tr>
<tr>
<td>Tomcat</td>
<td>8.0</td>
<td>[13]</td>
</tr>
<tr>
<td>Apache HTTP Server</td>
<td>2.4</td>
<td>[10]</td>
</tr>
<tr>
<td>PHP</td>
<td>5</td>
<td>[55]</td>
</tr>
<tr>
<td>Jersey</td>
<td>1.9</td>
<td>[46]</td>
</tr>
<tr>
<td>cURL</td>
<td>7.39</td>
<td>[20]</td>
</tr>
</tbody>
</table>

4.2 Implementation

In our prototype, we use HDFS and Hive as the big data distributed storage of the conceptual architecture, shown in Figure 4.2. HDFS and Hive use Sqoop as the ETL component in the BINARY architecture. YARN is used as the back-end resource manager. We include HCatalog, MapReduce and Hive as the engines and MySQL as the RDBMS components, respectively. We developed the related resource adapters to support the back-end resources in our prototype framework.
4.2.1 Web Server and User Interface Layer

Apache http server is selected as the web server. It hosts the User Interface layer, which contains our dynamic web pages (represented by PHP scripts in Figure 4.2) to provide easily accessible web interfaces to the end user. We developed these pages using PHP [55], an open-source scripting language processing engine that executes on the server side to render custom web pages.
The PHP application server runs on the web server and handles all the communications between the Applications layer and the dynamic web pages. Well-formed requests from the dynamic web pages that constitute the web interface layer of our framework are then forwarded to the designated components in the Application layer. The interactive connection to the appropriate back-end resources works through the RESTful API, discussed in section 3.1.2.1. For
each selection made by the end user, the PHP code generates a URL based on the user-selected information and sends it to the RESTful API for processing and displays the returned data or results.

The PHP application can access RESTful APIs using the Client URL Library (cURL) extension, which allows users to connect to and communicate with many different types of servers using many different types of protocols.

Since the framework is a prototype for ad-hoc query execution, the first screen, shown in Figure 4.3, displays the available data resources and requires that the user to select one or more of the data source(s) as needed from the list. A connection to each selected data resource gets established and the list of available tables including the resource names is shown to the user. In the case that one table is selected, a list of columns (attributes) of that table is fetched and shown to the user. It is possible to narrow down the presented data adding filters such as relational operators (equal, greater than, less than, etc.) and logical operations (AND, etc.). In the case that two tables are selected from the same or different resources, the data from these two tables which meet the user-defined conditions are joined and presented. The joined data is saved in the HDFS and can be used for look up or further join operations.
4.2.2 Java Application Server and Application Layer

The URLs generated by the PHP web pages are sent to the RESTful APIs in the Application layer including the required information (obtained from the end user) to call the appropriate functions in the selected resource adapter (see section 3.1.2.2). A list of calls in the RESTful API is given in Table 4.2.

Security is not an important consideration for our proof-of-concept implementation so the username and password are passed directly as part of the URL in the HTTP commands. For a more secure implementation, we could adopt another approach, such as creating a login service that takes the user’s credentials and returns a token. We then use that token instead of the username and the password in the communication. We can further assign expiration dates for the tokens, and manage this functionality on the server.

We developed the RESTful APIs and the resource adapters in Java. We use Apache Tomcat [13], which is a Java application server, as the web container for our RESTful APIs. We use JAX-RS/ Jersey, which is the Java API for RESTful web services and provides required reference for its implementation in a Java servlet container (see Appendix A). We implement each adapter class as a Java package and the RESTful interfaces as separate java classes in the same package. Each adapter is defined as a domain class and its RESTful interfaces are considered as the resource class. A resource class returns the required information from the related domain class using JAX-RS annotations (e.g., @PATH gives path to the application name @GET gets request for HTTP).

All the resource adapters and corresponding RESTful APIs are defined in a single Java project and packed together in a Web application Archive (WAR) file format [79] and deployed on the Java application server. The WAR file contains all the required Java components and simplifies verification and deployment of web applications. A single Java project enables calling
RESTful APIs for different resource adapters in a uniform way. Also, having a WAR file, a compiled web Java project with all the required libraries, provides the flexibility of having the Resource layer and the Application layer on separate machines.

We developed a Hive adapter to facilitate the integration of the different resources into HDFS and to enable data access and querying using HiveQL as presented in Figure 4.4. This approach provides an abstraction layer to allow a unified view of data regardless of their physical location.

Figure 4.4: HiveQL as a uniform language in the resource adapters.
<table>
<thead>
<tr>
<th>Resource Adapter</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hive</td>
<td>RetrieveColumn</td>
<td>Displays column’s (attributes) name in a given table</td>
</tr>
<tr>
<td></td>
<td>RetrieveDB</td>
<td>Returns available databases</td>
</tr>
<tr>
<td></td>
<td>RetrieveTables</td>
<td>Returns a list of tables in a given database</td>
</tr>
<tr>
<td></td>
<td>RetrieveTable</td>
<td>Shows the contents of a selected table</td>
</tr>
<tr>
<td></td>
<td>DescribeTable</td>
<td>Displays the information about a column in a given table</td>
</tr>
<tr>
<td></td>
<td>ShowSelectedTableColumns</td>
<td>Displays only the contents of selected columns in a given table</td>
</tr>
<tr>
<td></td>
<td>JoinTables</td>
<td>Joins two tables in Hive</td>
</tr>
<tr>
<td></td>
<td>JoinTablesWithMySQL</td>
<td>Joins a table from Hive with another table from MySQL</td>
</tr>
<tr>
<td>HCatalog</td>
<td>RetrieveDB</td>
<td>Returns a list of available databases existing in Hive Metastore</td>
</tr>
<tr>
<td></td>
<td>RetrieveTables</td>
<td>Returns a list of tables in a given database existing in Hive Metastore</td>
</tr>
<tr>
<td>Sqoop</td>
<td>SqoopImport</td>
<td>Imports a given table from a selected database in MySQL to HDFS</td>
</tr>
<tr>
<td></td>
<td>SqoopExport</td>
<td>Exports a given table from HDFS to a given database in MySQL</td>
</tr>
<tr>
<td>MySQL</td>
<td>RetrieveTables</td>
<td>Returns a list of tables in a given database</td>
</tr>
<tr>
<td></td>
<td>RetrieveTable</td>
<td>Shows the contents of a selected table</td>
</tr>
<tr>
<td></td>
<td>DescribeTable</td>
<td>Displays the information about a column in a given table</td>
</tr>
<tr>
<td></td>
<td>ShowSelectedTableColumns</td>
<td>Displays only the contents of selected columns in a given table</td>
</tr>
<tr>
<td></td>
<td>DescribeTableforHive</td>
<td>Returns the information about the column names and types in a given table</td>
</tr>
<tr>
<td></td>
<td>JoinTables</td>
<td>Joins two tables in MySQL</td>
</tr>
</tbody>
</table>
4.2.3 Back-end Resources

Back-end resources reside in the Resource layer of the conceptual architecture and consist of various engines and data sources as listed in section 4.2, which communicate with their corresponding adapters in the Application layer. The back-end resources are typically distributed on multiple servers or virtual machines.

As mentioned in section 2.1.2, YARN runs on top of the HDFS as a resource management operating system and enables the engines to be executed on top of it (see Figure 2.2). Sqoop is a component of the architecture that supports bulk data transfer between HDFS and MySQL [50]. MySQL, a popular open-source relational database management system, is the RDBMS resource provided in our prototype.

In the prototype implementation of BINARY Hadoop MapReduce, Hive, and Sqoop are installed to run on Apache YARN as shown in Figure 4.2. These engines, along with engines outside Hadoop, such as MySQL are connected to the resource adapters in the network shown in Figure 3.3.

4.3 Functionality

A high level overview of the workflow is presented in Figure 4.5. Users are initially presented with a web page providing the option of choosing from three different resources: one or both of Hive and MySQL, or HCatalog. The dynamic web pages written in PHP retrieve necessary data to render on the web pages by calling the appropriate REST API. For example, if a user selects HCatalog, all the tables in Hive metastore are listed on the next web page. This operation represents the capability of accessing data from the Hive metastore through HCatalog. When the user selects MySQL or Hive (or both), all the tables inside the selected resource(s) are listed on the next page (as resource.table name). Now if a user selects a single table, the framework assumes that the user wants to observe the contents of that table. As a result, on the next web page the user is given the choice of selecting the desired attributes to retrieve and
setting the filter conditions. The results are then retrieved accordingly and displayed in a tabular format on the screen. In the case of selecting two or more tables, it is assumed that the user wants to join those tables. Thus on the subsequent pages the user is given the choice to add or remove attributes to retrieve and join and/or filter conditions and the system then retrieves the data and displays the results.

The MySQL adapter includes standard functions such as “describe” to show the attributes of a table and their types, and “show” to list available tables in a selected database. There are also custom functions, for example “select” to select specific attributes of a table with or without filtration. Also, a “join” function is written to join two selected tables from MySQL and save the results as a table in MySQL (shown to the user afterwards). We developed parsers for MySQL adapter to read the query received as a string from the REST API or PHP, to clean it (e.g. removing extra comma), and to figure out whether the user wants to see the contents of a table or to join two tables. The results of a join function can be used in the subsequent join function.

Figure 4.5: Web interface workflow.
Functions of the Hive adapter are at the core of our proposed framework since they allow data integration and summarization for large datasets. These functions include standard SQL-like functions for creating or dropping a table, describing or selecting a table, showing databases, and counting rows of a table. We developed a custom function to handle the join operation on tables either both from Hive or one from Hive and the other from MySQL. In the case one table is from Hive and the other is from MySQL, we need to create a table in Hive that is identical to the MySQL table. We execute the following steps for this purpose (in addition to parsing as mentioned above):

- First, we need to recognize which table is from MySQL, then call the MySQL adapter from inside the Hive adapter to retrieve the names and the types of the table attributes by calling the ‘describe’ function. The Sqoop adapter is called inside the Hive adapter to import the MySQL table into HDFS. Then the Hive adapter is used to load data from HDFS to the new table created in Hive which is identical to the MySQL table.
- Next, the join function in the Hive adapter is called to carry out the rest of the join operation. Results of a join are stored and can be used in subsequent operations, for example a join with a third table.

We modified the Sqoop client API in the adapter to import data from MySQL into HDFS (in Sqoop v.1.99.3, as shown in Figure 4.6). This is done by passing parameters such as the type of job (import/export), schema name, table name, and output directory name within a path. Sqoop refuses to import tables with the name already existing in HDFS and hence we added a timestamp at the end of each table name to define unique table names.
For reliability reasons, we used Sqoop v.1.4.4 to directly import data from MySQL to Hive which is discussed further in section 4.4. In the proposed software solution data is transferred sequentially in a form of table from MySQL to the default database of Hive via Sqoop v.1.4.4. For example;

```
sqoop import --connect jdbc:mysql://localhost/cpcssn --username root -P --table Patient
-m 1 --hive-import
```

is executed to import a Patient table from MySQL to Hive directly as shown in Figure 4.6.

By default Hive is configured to use Derby as the metastore, which supports only one active user at a time. Choosing to replace Derby with MySQL is a popular option for resolving the one user issue.

HCatalog has a RESTful interface via WebHCat (see section 2.2.2.2). Since we want to have a unified address for all the adapters, we used Apache HttpClient [11] in our HCatalog adapter to simulate the web browser in a java class for HCatalog to transmit and receive HTTP
messages. Our provided parser only keeps the required information and removes the unnecessary input.

4.4 Limitations

To have a high degree of flexibility, our proof-of-concept prototype is designed such that the User Interface and the Application layer of our framework can reside on any machine and allow the user to connect remotely to the server hosting the data resources. This approach provides advantages such as extendibility and independence from the hardware and operating system platforms. For this reason, we use Sqoop version 1.99.3 in our Sqoop adapter design, which has a Sqoop2 server to enable Java code to remotely access and import any table from MySQL into HDFS. The code is validated by running it on a local machine and connecting to the databases on a cloud server. In practice, we observe that importing a number of attributes even with 100,000 rows from a MySQL table was successful. However, importing the complete table failed due to the unstable Sqoop version. The problem was resolved when we used the stable version of Sqoop (version 1.4.4), which has no server. The consequence of having no server is that the Java code can be used to invoke Sqoop only from the command line. Therefore, the Java code mentioned in section 4.2.2 has to be located on the same machine as the data resources. In the Sqoop adapter that we developed for the prototype, we implemented the functionality for data import and left the export function as a future work.

In the current version of our framework the logical operation ‘AND’ is included but we have not implemented the aggregation and sorting functions. Also, we have chosen that only the first thousand lines of each selected table or the result of a join operation are displayed on the web page. It is straightforward to show the whole result using a loop.
4.5 Summary

In this chapter, we described the implementation details of the proof-of-concept prototype of our proposed BINARY framework for executing ad-hoc queries involving different types of distributed data sources. We used a preferred set of existing big data tools, storage systems, and RDBMS software based on their functionality, flexibility, adaptability, and popularity to provide a seamless functionality in our framework (e.g. HDFS, MapReduce, YARN, Hive, HCatalog, Sqoop and MySQL). BINARY is extendable to other big data resources (e.g. HBase, Cassandra) and RDBMSs (e.g. PostgreSQL, or Oracle via JDBC).

We explain our three-layer architecture consisting of a User interface layer, Application Layer and Resource layer of which the first two are hosted on a common machine and help connect to the distributed back-end resources hosted on the Resource layer, possibly residing on another computer. We developed dynamic web pages to provide access to the back-end resources to perform ad-hoc query.

BINARY not only supports ad-hoc queries on remote distributed data sources but also allows integration between RDBMSs and non-relational databases to query data from multiple different data sources via web interface and RESTful API.
Chapter 5

VALIDATION

In this chapter we present some test case scenarios to validate the functionality of the BINARY prototype and to provide insight into its operation. We illustrate how the framework can be used for ad-hoc query by consolidating data from different distributed sources into a big data store, supporting for various data sources and providing a user-friendly web interface to data access.

5.1 CPCSSN Data

Data from the Canadian Primary Care Sentinel Surveillance Network (CPCSSN) [23] is used in our validation under a non-disclosure agreement. The extracted data contains anonymized data of patients with the following chronic diseases: Chronic Obstructive Lung Disease, Depression, Diabetes, Hypertension, Osteoarthritis, and Chronic Neurological Disease including Dementia, Epilepsy, and Parkinson's disease. CPCSSN data is used for primary care research using Emergency Medical Records (EMR) in 8 provinces in Canada. We used this data to predict patients who need emergency care before being hospitalized in order to decrease health care expenses. Data from Kingston, Ontario and the regions around it is used to test our prototype with 26 entities having in the range of 3 to 26 attributes and number of records from 10 to more than 1 million.

5.2 Test Case Scenarios

Ad-hoc querying is the user exploring the data sources, discovering what tables and attributes are available and then issuing a series of queries to find out about the data. The queries could be on a single table with one or more filtering conditions. They could also join the data
from different tables applying filtering conditions on the joined data. Ad-hoc queries are unpredictable and exploratory.

5.2.1 Scenario: Querying a table from Hive or MySQL

Our framework provides uniform web interfaces for querying data from different types of back-end data sources. We illustrate in Figure 5.1 the scenario where a user queries data from a table in a selected data source. The user can browse attributes and add or remove the required ones by filtration as shown in Figure 5.2 with no need to know where the location of Hive is or to use any programming. The user can see contents of a Hive table as in Figure 5.3 with all the available fields in the table.

![Figure 5.1: Selecting a table in Hive.](image)
The user can query the contents of a table MySQL and any subset of fields can be selected for visualization as presented in Figure 5.4 and Figure 5.5.
5.2.2 Scenario: Showing tables using HCatalog:

Our framework provides a web interface for the user to observe all the tables available in the Hive metastore and for other data processing tools like Pig and MapReduce via HCatalog as shown in Figure 5.6 and discussed in section 2.2.1.2. In our prototype we installed only the Hive
metastore which can be accessed via HCatalog but the implementation can be extended for other tools. We implemented this feature as a provision for future work to add support for more data processing tools in our framework.

Figure 5.6: List tables in Hive metastore via HCatalog.

5.2.3 Scenario: Join query

One of the primary goals of our work is to provide a unified view of data from multiple data sources. Using the web interface of our framework the user can browse and select desirable attributes and conditions on data from different sources and then join the data into a single unified view. The selected attributes or conditions can be modified before submission. The result of the join operation is shown to the user. Some examples of joining two tables are provided in this section. More specifically:

- Joining two tables both from Hive is provided in Figure 5.7. The joined table is given in Figure 5.8.
- Joining two tables from MySQL is depicted in Figure 5.9. The result is shown in Figure 5.10.
- Joining two tables one from Hive and the other from MySQL is depicted in Figure 5.11 and the outcome is illustrated in Figure 5.12.
5.2.4 Scenario: Consolidate legacy and recent data from two different types of data sources

Suppose that initially data about patient visits to family doctors was stored in a MySQL table. Recently, considering the large scale of datasets, the data is managed using big data tools such as Hadoop and Hive as a query engine. In these cases, a patient’s visit is recorded in Hive tables. The ad-hoc query builds a unified view of patient visits from the legacy and recent data regardless of their resources.

Figure 5.7: Join between two tables in Hive.
Figure 5.8: Result of a join between two tables in Hive.

Figure 5.9: Join between two tables in MySQL.
Figure 5.10: Result of a join between two tables in MySQL.

<table>
<thead>
<tr>
<th>BirthYear</th>
<th>Sex</th>
<th>Name_orig</th>
</tr>
</thead>
<tbody>
<tr>
<td>1962</td>
<td>Female</td>
<td>Gen List Consult - Dermatology (02)</td>
</tr>
<tr>
<td>1946</td>
<td>Female</td>
<td>Gen List Consult - Resp disease (47)</td>
</tr>
<tr>
<td>1933</td>
<td>Male</td>
<td>Gen List Consult - General surgery (03)</td>
</tr>
<tr>
<td>1933</td>
<td>Male</td>
<td>Gen List Consult - Hematology (61)</td>
</tr>
<tr>
<td>1933</td>
<td>Male</td>
<td>Gen List Repeat consult - Cardiology</td>
</tr>
<tr>
<td>1940</td>
<td>Female</td>
<td>Gen List Consult - Orthopedics (06)</td>
</tr>
<tr>
<td>1948</td>
<td>Female</td>
<td>General List Consult Radiation Oncology (34)</td>
</tr>
<tr>
<td>1989</td>
<td>Female</td>
<td>Gen List Consult* - OB/Gyn (20)</td>
</tr>
<tr>
<td>1934</td>
<td>Male</td>
<td>Gen List Consult - General surgery (03)</td>
</tr>
<tr>
<td>1932</td>
<td>Female</td>
<td>Gen List Consult* - OB/Gyn (20)</td>
</tr>
<tr>
<td>1932</td>
<td>Female</td>
<td>Internist Medicine Partial Assessment</td>
</tr>
<tr>
<td>1999</td>
<td>Male</td>
<td>Gen List Consult - Pediatrics (26)</td>
</tr>
<tr>
<td>1938</td>
<td>Female</td>
<td>Gen List Consult - Neurology (04)</td>
</tr>
<tr>
<td>1943</td>
<td>Female</td>
<td>Gen List Consult - Ophthalmology (23)</td>
</tr>
<tr>
<td>1957</td>
<td>Female</td>
<td>Gen List Consult - Neurology (04)</td>
</tr>
<tr>
<td>1957</td>
<td>Female</td>
<td>Gen List Consult - Orthopedics (06)</td>
</tr>
<tr>
<td>1993</td>
<td>Male</td>
<td>Gen List Consult - Pediatrics (26)</td>
</tr>
<tr>
<td>1927</td>
<td>Male</td>
<td>Gen List Repeat consult - Anesthesiology (01)</td>
</tr>
<tr>
<td>1972</td>
<td>Female</td>
<td>Gen List Consult - General surgery (03)</td>
</tr>
<tr>
<td>1948</td>
<td>Male</td>
<td>Gen List Repeat consult - Cardiology</td>
</tr>
<tr>
<td>1948</td>
<td>Male</td>
<td>Gen List Repeat consult - Anesthesiology (01)</td>
</tr>
<tr>
<td>1965</td>
<td>Female</td>
<td>Gen List Consult - Neurology (04)</td>
</tr>
<tr>
<td>2001</td>
<td>Male</td>
<td>Gen List Consult - Orthopedics (06)</td>
</tr>
</tbody>
</table>
Figure 5.11: Join between two tables in Hive and MySQL.

Figure 5.12: Result of a join between two tables in Hive and MySQL.
### 5.2.5 Scenario: Repetitive Join Operations

The user can also merge specific attributes of two tables in Hive one of which being the result of a previous join operation. The outcome is shown in Figure 5.13. The user is able to retrieve the information about disease of patients from the first join and then use it in the second one to retrieve the relation among disease, sex and age.

![Figure 5.13: Result of a join between two tables in Hive.](image)

### 5.3 Summary

The functionality of the BINARY prototype, and its suitability for ad-hoc querying over big data, are successfully validated with the help of CPCSSN health care data. The test case scenarios demonstrate how the framework is used to merge data from Hive or MySQL tables in HDFS in a repetitive join operation. Hive is known as a scalable big data storage for analyzing
and querying large amounts of batch data. In these tests we imported the selected data from MySQL to HDFS and then performed the join operation using Hive. The test cases demonstrate the capability of querying a table in Hive or MySQL and showing tables through HCatalog. The user can perform all these operations just by selecting the options offered in the web pages.
Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

With the enormous growth of data and the availability of low-cost and reliable storage, enterprises can no longer rely only on RDBMSs to maintain their data. They are shifting toward big data paradigms to handle a wider variety of data types with more flexibility and at the same time have access to RDBMS sources. In many instances, the information derived from large scale and unstructured data in raw format cannot be effectively understood unless it is processed alongside structured data. Hence, having access to both the big distributed data sources and the traditional relational databases is essential for the purpose of ad-hoc querying and data analytics.

The current state-of-the-art big data systems have several shortcomings in this regard: they do not necessarily support hybrid databases consisting of both scalable big data storage and RDBMS types and their data integration; they do not have a universal view for query and manipulation of various data types and resources through web access, and they often rely on proprietary data formats, codes and hardware.

In this research, we present the BINARY framework that has the following properties:

- it integrates data from multiple resources into HDFS on the back-end using Apache Hive;
- it provides RESTful APIs for all the data resources to facilitate independent updating of the front-end programs without affecting the back-end data infrastructure; and
- it provides a web interface to the end users to access and join data between RDBMS and HDFS with no requirement of programming knowledge or installation of any software.

Ad-hoc analysis, which extracts insights from historical trends and finds correlations
between data stored in multiple data sources, requires capabilities for big data such as providing access to large data resources, maintaining speed of access for scalable data, supporting various data types of data sources and enabling flexible, ubiquitous data access and query. BINARY satisfies the requirements of big data ad-hoc querying. It has a multi-layer SaaS architecture including: the User Interface layer as a connection between the end users and the Application layer; the Application layer to connect the User Interface and the Resource layers, which includes a set of RESTful APIs; and the Resource layer that contains ETL tools, analytics and storage engines as well as RDBMSs.

A proof-of-concept prototype implementation of BINARY has a web-based user interface using PHP for convenient remote access. Through this interface the end users can perform their queries and join tables from various resources as well as visualize the data in tabular formats. The prototype is implemented on the OpenStack cloud system running Linux. We use HDFS and Hive as big data distributed and scalable storages. HDFS and Hive use Sqoop to transfer bulk data. YARN is used as Back-end resource manager and HCatalog, MapReduce and MySQL as the data resources. We developed the related resource adapters to support the back-end resources in our prototype framework. The BINARY has the flexibility of hosting the User Interface and Application layers on any server hardware and then remotely accessing the back-end resources on the Resource layer located on other machines.

The functionality of the BINARY prototype is validated with the help of a sample health care data. Several ad-hoc querying test case scenarios are presented as part of the validation.

6.2 Future Work

The current implementation of BINARY initially used Sqoop version 1.99.3 with Sqoop2 server to enable our Java code to remotely access and import any table from MySQL to HDFS. Importing a subset of a table attributes from a MySQL table with more than 100,000 rows was
successfully verified. However importing the full table faced challenges due to the instability of this version of Sqoop. Using the stable version of Sqoop (version 1.4.4) resolved the aforementioned issue at the cost of having no server and subsequently the User Interface and Application layers had to reside on the same machine where the data resources are located. The system is fully functional however the full advantage of architecture is not used which is the flexibility of having the Application layer and the Resource layer on different machines. Also, the current prototype has a limited number of logical operations and no aggregation and sorting functionalities. In addition, only the first thousand rows of each selected table (or the result of a join operation) are displayed on the web page. The current implementation of BINARY does not provide the export functionality of the Sqoop adapter.

The proposed BINARY framework can be extended to other data processing engines beyond MapReduce such as Apache Tez. Tez provides Directed Acyclic Graph (DAG) execution framework that gives a more natural model for execution plan in query execution systems like Hive. Tez improves the Hive query performance by providing fast response times.

BINARY can be expanded to operate with commercial data sources of RDBMS and HDFS categories. It is possible to add the Apache HBase engine to utilize unstructured data and perform analytics. Other options are Cassandra and Mongo DB. BINARY can be extended to use HCatalog with Apache Pig and Hive, in order to provide an abstract view of data stored via those engines. Extension of the BINARY to a multi-user application deployed on the cloud is another possibility.

Apache Drill is an open-source ‘interactive’ SQL query engine for Hadoop that provides fast interactive queries for large datasets. At the time of this research, a shortcoming of Drill is the lack of a connector to RDBMS. On the other hand, the extensible architecture of Drill allows storage plugins for specific requirements to be implemented with RESTful API [5]. Our software solution can be used as a storage plugin for Apache Drill to establish connectivity to RDBMS.
Bibliography


[61] Scott C. Gray, Fatma Ozcan, Hebert Pereyra, Bert van der Linden and Adriana Zubiri. 2014. “SQL-on-Hadoop without Compromise.”


“VirtualBox.” https://www.virtualbox.org/.


Appendix A
RESTful Setup

For our prototype implementation, Jersey is added to the project classpath, with Apache Tomcat 8.0 as a web container on the Windows platform. The servlet is configured via the `web.xml` as presented in Figure A.1. It analyzes the incoming HTTP request and scans certain packages to select the correct class and method to respond to the request. The base URL of the servlet is

http://your_domain:port/display-name/url-pattern/path_from_rest_class

```
Web.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <display-name>Restful Web Application</display-name>
  <servlet>
    <servlet-name>storage-servlet</servlet-name>
    <servlet-class>
      com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <init-param>
      <param-name>com.sun.jersey.config.property.packages</param-name>
      <param-value>ca.queensu.cs</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>storage-servlet</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Figure A.1: Jersey servlet configuration.
In REST the location of the resources for RESTful services can be specified via URLs. The client issues a URL to locate the resources and the request a service from the server.

In Java API for RESTful Web Services (JAX-RS), we use @Path annotations to the Java class files or the Java methods to define the relative URL of the resource. This annotation can be used to define resources through JAX-RS subresource locators and subresource methods. The structure of URLs used in our prototype system follows the pattern below:

http://DomainName:PortNumber/ProjectName/rest/ResourceAdaptorName/FunctionName?Parameter1=Value1&Parameter2=Value2&…&ParameterN=ValueN

As an example in our implementation, DomainName is “localhost”, PortNumber is 8080, and ProjectName is HadoopREST where the entire resource adapter classes are defined. The next portion of the URL is always “/rest/” which is the full servlet URL pattern (see Figure A.1). Then ResourceAdaptorName is one of the resource adaptor names from Table 4.2 and FunctionName is one of the functions specific to the selected adaptor again from Table 4.2. Then the parameters specific for that function and their values are listed. The common parameters include the IP of the server, which has the data resource on it, the user name and password for authentication purposes. The parameter portion of the URL is function specific (regardless of IP and authentications) and vary based on the given ad-query. For instance, a URL can be as:

http://localhost:8080/HadoopREST/rest/hive/retrieveDB?ip=130.15.6.122&userName=root&passWd=myPassWd)