ABSTRACT

Boggaram Gopinath, Chandra Mohan, Efficient In-Database Analytics through Embedding MySQL into R. (Under the direction of Professor Nagiza F. Samatova).

High-performance analytics of data at extreme scales is a well-recognized challenge by both scientific and business communities. The goal of this Master’s thesis is to explore effective and efficient ways of performing statistical analysis of the data stored in large-scale relational databases (DB). The underlying hypothesis is that in-database analytics offers a plausible solution to this challenge by coupling analytical and database capabilities together. Such a coupling may allow analytical workflows to be executed without moving the data out of the databases and therefore avoid transferring the data over the network. Therefore, in-database analytics may potentially reduce the overall latency, assure better data governance and security, and enable analytical solutions to scale to larger data sets with more efficient resource utilization.

In-database analytics can be realized through the following complementary approaches: (a) analytics-in-DB places analytical workflows inside a DB server and (b) DB-in-analytics embeds the DB server into the memory space of analytical routines. The former has been primarily driven by the database community through various mechanisms, such as user defined functions, stored procedures, compiled codes, etc. The latter is an emerging approach dominated by open source, robust, and scalable solutions in their infancy.

The focus of this Master’s thesis is on developing an open source and efficient in-database analytics solution via embedding a MySQL server into an R statistical data analysis environment. To the best of our knowledge, this is the first study that integrates analytical capabilities of R with a MySQL database management system in an embedded manner. Specifically, the three novel ways for embedded DB-in-analytics are proposed and systematically evaluated. In contrast to existing wrapper-based approaches that provide wrapper APIs to MySQL functions, the proposed embedded solutions improve the time efficiency of R’s access to the MySQL DB by 650% to 1900%.
Efficient In-Database Analytics through Embedding MySQL into R

by
Chandra Mohan Boggaram Gopinath

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science
Raleigh, North Carolina
2008

APPROVED BY:

Dr. Steffen Heber

Dr. Kemafor Anyanwu

Dr. Nagiza F. Samatova
Chair of Advisory Committee
DEDICATION

Dedicated to all those who made this thesis possible directly or indirectly.
**BIOGRAPHY**

Chandra Mohan BG received his B.E. in Information Science and Engineering from J.S.S Academy of Technical Education, Bangalore affiliated to Visweswaraiah Technological University in 2004. He worked as a research assistant under Dr.V.V.S.Sarma, Computer Science and Automation, Indian Institute of Science from May 2005 to May 2006. He has identified some of the major novel conceptual contributions of Indian Logic and has shown its applicability to Knowledge Representation in Computer Science. Pursuing his interest in languages and computer science he enrolled to the Dept. of Computer Science, North Carolina State University on August 2006. He worked as a research assistant under Dr.Nagiza F.Samatova under the project “In- database Analytics with R”. He is expected to graduate on October 2008 and will be joining Microsoft Corporation, Redmond, USA.
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Nagiza F. Samatova for guiding me in solving problems with her excellent problem analysis skills and effective problem-solving strategies. I would like to thank Dr. Kemafor Anyanwu and Dr. Steffen Heber for agreeing to be my committee members and also for their valuable comments and suggestions on my thesis. I would like to thank my sister Lakshmi B.G. for her insights into solving problems during difficult times of thesis work. I would like to thank Paul Breimyer for his constant support, help, and motivation throughout my thesis work. I would also like to thank Guruprasad Kora for introducing me into this project. I would like to thank all those who financially supported me to complete this thesis: my advisor, parents, and part-time employer. I would like to thank my friends Sundar Subramanyam E.K., Harsha Girish, Chaya Narayan Kutty, and Aaditya Dalniya for motivating and supporting me to pursue thesis work.

Above all, I am highly grateful to Dr. David Thuente, DGP, Dept of Computer Science. Dr. Thuente was always there to guide me by patiently answering my questions with a smiling face. Without the guidance of Dr. Thuente, I definitely would not have completed my graduate studies smoothly.
# TABLE OF CONTENTS

LIST OF TABLES .................................................................................. vii

LIST OF FIGURES ........................................................................... viii

1 Introduction .................................................................................. 1
   1.1 Motivation ............................................................................ 1

2 Problem Statement ....................................................................... 7
   2.1 Motivation ............................................................................ 7
   2.2 Specific Goals ....................................................................... 8
      2.2.1 RMySQL Scenario .......................................................... 8
      2.2.2 BridgeRMySQL Scenario .............................................. 9
      2.2.3 BridgeRMySQLD Scenario .......................................... 11
      2.2.4 BridgeRMySQLDMap Scenario .................................... 13
   2.3 Major Contributions .............................................................. 14

3 Design and Architecture .............................................................. 16
   3.1 High-level architecture of the system ..................................... 16
   3.2 Design of the RMySQL Scenario .......................................... 19
   3.3 Design of the BridgeRMySQL scenario ................................. 21
   3.4 Design of the BridgeRMySQLD scenario .............................. 23
   3.5 Design of the BridgeRMySQLDMap Scenario ....................... 25

4 Evaluation .................................................................................. 31
   4.1 Cost evaluation parameters for the In-database analytics with R . 31
   4.2 Test Results .......................................................................... 32
   4.3 Comparative analysis of the scenarios .................................... 36
   4.4 Test Environment Setup ...................................................... 37
   4.5 Test Environment Configuration .......................................... 38

5 Conclusion and Future Work ....................................................... 39
   5.1 Conclusion .......................................................................... 39
   5.2 Future Work ........................................................................ 40

Bibliography ................................................................................. 42

A MySQL Installation .................................................................. 45
   A.1 Installing MySQL 4.1.22 ....................................................... 45
   A.2 Embedded MySQL shared library ........................................... 48
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Table Legend</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>RMySQL Performance Table</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>BridgeRMySQL Performance Table</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>BridgeRMySQLD Performance Table</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>BridgeRMySQLDMap Performance Table</td>
<td>33</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1 Susy is questioning how to access data smartly? ................................................. 2
Figure 1.2 Embedding R’s Analytical routines into MySQL....................................................... 4
Figure 1.3 Embedding MySQL into R ....................................................................................... 5
Figure 2.1 RMySQL Scenario: State-of-the-art approach .......................................................... 9
Figure 2.2 BridgeRMySQL Approach ..................................................................................... 11
Figure 2.3 BridgeRMySQLD Scenario ................................................................................... 12
Figure 2.4 RMySQLDMap Scenario ....................................................................................... 14
Figure 3.1 High-level architecture of the RMySQL/BridgeRMySQL scenarios involving stand-alone MySQL server ................................................................. 17
Figure 3.2 High-level architecture of the BridgeRMySQLD/BridgeRMySQLDMap scenarios involving Embedded MySQL server ......................................................... 18
Figure 3.3 RMySQL Design and Architecture ..................................................................... 22
Figure 3.4 BridgeRMySQL Architecture .............................................................................. 24
Figure 3.5 BridgeRMySQLD Architecture ......................................................................... 26
Figure 3.6 BridgeRMySQLDMap Architecture ................................................................... 28
Figure 3.7 Design of BridgeRMySQLDMap Package ............................................................ 30
Figure 4.1 Comparison of time to query server in secs ......................................................... 34
Figure 4.2 Comparison of time to fetch data in secs .............................................................. 34
Figure 4.3 Graph Legend ........................................................................................................ 34
Figure 4.4 Comparison of time to access data in secs ........................................................... 35
Figure 4.5 Comparison of time to typecast data in secs ....................................................... 35
Figure 4.6 Graph Legend ......................................................... 35
Figure 4.7 Comparing Stand-alone and Embedded server ..................... 37
Figure E.1DBI Package .......................................................... 68
Figure E.2RMMySQL Package ................................................ 71
Chapter 1

Introduction

1.1 Motivation

Business Intelligence (BI) [24] is a term introduced by Howard Dresner of the Gartner Group in 1989 to describe a set of concepts and methods to improve business decision making by using fact-based support systems. SAS, Microstrategy, Oracle, IBM, Microsoft and many other companies have come out with business intelligence products that are data-driven Decision Support Systems (DSS).

BI systems usually gather large volumes of data over years into data warehouses and data marts. BI products need to access these large volumes of data for information extraction, analysis, and reporting. As data sizes grow, these systems often find it difficult to access large volumes of data back and forth from data warehouses and data marts to data analytical engines. Transfer of large volumes of data back-and-forth consumes a lot of resources in terms of network bandwidth, transfer time, and memory space. Transfer of data also leads to data redundancy. This large amount of traffic also gives rise to data governance issues in order to make sure that it is not accessible by third party. Sometimes the data needs to be prepared for analysis; such data processing also consumes a lot of resources. These issues are gaining importance as the size of the data is growing. So the question is whether there are ways to access these large volumes of data smartly to analyze them effectively (Figure 1.1)?
In-Database Analytics Technology offers a way to solve the problem by bringing the data analysis tools closer to databases; this may result in efficient resource utilization and increased productivity in seeking answers to questions from data. It enables analytical work flows to be embedded into a database. These work flows can then be executed without moving the data out of the database. Such an approach may result in moving less data and reducing data latency and decision latency. It may also scale the solutions to larger databases with little or no data governance issues, thus leading to overall performance gain for the system. Two complementary approaches to bring analytical capabilities and databases together are the following:

- **Analytics-in-DB**: Place the analytical routines and work flows inside a database server, running analytical routines as an independent process so that analytical routines can access data easily. Here the data-analysis part of the client functionality is placed inside the database server. In this case, memory may not be an issue since analytical routines often analyze the subset of the data in large databases and the size of analytical routines is likely to be small compared to the size of the data in the large databases.
• **DB-in-Analytics**: Place the database server inside the analytical routines space so that analytical routines can easily access the data. Here, the database server functionality is executed in the client’s memory space. In this case, memory may be an issue since the database memory requirements may be relatively large compared to the available client’s memory space.

**R** [7] is an integrated suite of software facilities for statistical analysis, data manipulation, calculation, and graphical display. **R** is an implementation of the **S** language that was developed at Bell Laboratories by Rick Becker, John Chambers and Allan Wilks, and also forms the basis of the **S-PLUS** systems. **R** is extensible, widely used in academia, and is becoming more common in industry[14]. **R** is licensed under GNU General Public License(GPL)[13]. **R** supports various platforms including Linux, Macintosh, and Windows. **RGui** is a graphical interface for **R** and it is also available in command-line driven mode.

**MySQL** has been identified as one of the world’s most popular open source databases and is the fastest growing database system in the industry. **MySQL AB** is reporting over 8 million active installations and nearly 50,000 downloads per day. **MySQL** is rapidly becoming the database system of choice for system integrators. According to an article in **SD Times**, **MySQL** is now the number three “Top Deployed Database” in a recent survey of over 900 readers.

The broader goal of this thesis work is to leverage powerful **In-database Technology** to integrate statistical capabilities supported by **R** with the database management capabilities supported by **MySQL**.

*Analytics-in-DB*, illustrated in Figure [1.2] can be achieved in any one of the following ways:

- Add analytical routines as **MySQL User Defined Functions** to the **MySQL** server.
- Add analytical routines as **MySQL Native functions** to the **MySQL** server.
- Add analytical routines as new **SQL commands** to the **MySQL** server.
- Add analytical routines as **MySQL stored procedures** to the **MySQL** server.
Figure 1.2: Embedding R’s Analytical routines into MySQL

- Add analytical routines as **compiled source code** to the MySQL server.

One can invoke any of the above analytical functions from the MySQL prompt.

**DB-in-Analytics** illustrated in Figure [1.3](#) can be achieved by:

- Embed the embedded version of MySQL database server in the R’s client memory space.

The state-of-the-art in **In-database Analytics** technology shows that the current trend in industry is **Analytics-in-DB** due to the need for data analysis by large database vendors, such as Teradata. The emerging **DB-in-Analytics** field will likely be the **future trend** as more and more business intelligence vendors, such as SAS or Microstrategy, will start empowering their BI products with database management system capabilities. Hence, this thesis focuses on the **DB-in-Analytics**.

The major contributions of this thesis are the following:

- Explored the current ways to access data by the analytical capabilities of R.

- Proposed new ways of integrating analytical capabilities provided by R and MySQL, using the **DB-in-Analytics** approach.
This thesis proposes three novel ways of achieving DB-in-Analytics, by overcoming the shortcomings of existing approaches, such as RMySQL (see Section 2.2.1). The proposed new ways were tested and demonstrated the improvement in performance for the DB-in-Analytics approach with R and MySQL. The results show that access to MySQL by R has been improved by 650% to 1944% through the proposed ways of integration. The In-database Analytics has greatly reduced the network bandwidth requirement involved in transferring the data. The closer integration of database and analytical capabilities have reduced the transfer time to a great extent, which has resulted in faster data access, thus leading to better decision making to achieve optimized results.

This thesis is organized into the following chapters. The first chapter gives an introduction to In-database analytics with R and MySQL. The second chapter describes the problem statement. The third chapter describes the design and architecture of the proposed solution. The fourth chapter describes the experimental setup and results. The last chapter concludes the thesis and discusses possible future directions. Appendix A gives instructions on how to install MySQL-4.1.22 on Linux. Appendix B describes Embedded MySQL APIs.
E gives an overview of R database packages. Appendix C gives a BridgeR C/C++ example. Appendix D gives the implementation details of proposed solution.
Chapter 2

Problem Statement

2.1 Motivation

R is an integrated suite of software facilities for statistical data analysis, data manipulation, calculation, and graphical display. It has a large, coherent, integrated collection of intermediate tools for statistical data analysis. R [7] is an environment, in which many classical and modern statistical techniques have been implemented. R can perform diverse statistical analysis tasks, such as linear regression, classical statistics tests, time-series analysis, and clustering. Some of the core modules are supplied as base R environment. Due to R’s rapid development capabilities, R has been extended by a large collection of user packages.

R packages often find it difficult to access large volumes of data back-and-forth from MySQL in order to perform statistical analysis on these data. Transfer of large volumes of data from R to MySQL and vice-versa has a number of limitations: (a) large amounts of network bandwidth, transfer time, and memory space requirements; (b) concern with data governance issues to make sure that it is not accessible by the third party; and (c) data redundancy. Sometimes preparation of the data before its transfer from MySQL to R can also consume a lot of resources.

The aforementioned issues are not very critical, when the sizes of the databases and the transferred data are relatively small. These issues are becoming more critical as the size of the databases are becoming larger. In-Database Analytics Technology provides a way
to address the above issues by bringing the R and MySQL closer to each other for more efficient resource utilization and increased productivity in seeking answers to questions from data. There are a number of advantages in integrating R functionalities with the functionalities of MySQL, including but are not limited to:

1. **Data Sharing**: Data and data analysis results can be shared by both R and MySQL. Data sharing also leads to low network bandwidth consumption.

2. **Scalability**: Statistical analysis is performed by dedicated database management systems that can handle large data sets.

3. **Data Security**: Data does not need to be moved from the database and access to the data is managed by the database’s security mechanisms and policies.

4. **Data Fidelity**: Transferring data between different database repositories can result in data staling as part of the transfer process. *In-database Analytics* technology prevents data staling by avoiding the data transfer.

### 2.2 Specific Goals

Section 2.2.1 describes which R packages currently provide access to databases. Further sections proposes the three novel scenarios, each bringing the R and MySQL closer to each other.

#### 2.2.1 RMySQL Scenario

Currently, there are a few published/unpublished packages available for R to interact with databases. R/S-Plus Database Interface (DBI) [10] is a package that has defined all the base virtual classes for establishing database connection. Individual database drivers such as ODBC, Oracle, PostgreSQL, MySQL, etc., have extended this base class, accordingly, and have created packages RMySQL, RJDBC, RSQILITE etc., to access their corresponding databases. More details about the DBI and other packages and their object-oriented design and functionalities are described in Appendix E.
The current state-of-the-art approach is illustrated in the Figure 2.1, in which the R client accesses a stand-alone MySQL server through RMySQL package built on top of the DBI package. Data is transferred back-and-forth between the client and the server over the Wide Area Network WAN using the TCP/IP stack. R Data frames are used to temporarily store the data that is brought from the database. Since R Data frames do not support the data types supported by the databases, some information about the data from the databases might be lost when the data is mapped to/from the R Data frames.

2.2.2 BridgeRMySQL Scenario

R is a scripting language. Though R provides good abstractions for efficient data handling, R’s interpretable nature prevents from getting the desired performance. R provides a simple interface to load the RMySQL package into R environment and to call external C/C++ functions defined in the RMySQL package. However, R introduces overhead that is often higher than the time required to execute the underlying compiled C/C++ functions.

The following example briefly describes the overhead introduced by R in calling external C/C++ functions. According to R, any external user function cannot return any type of objects. All the changes to R have to be indicated through updating the variables, which are passed as arguments to a function. Additionally, R introduces overhead both before and after calling the external methods caused by copying and casting parameter variables.
The following C/C++ function defines a function with two input parameters.

```c
void function(int x, int y)
{
    //.....
}
```

The above function can be called from R using:

```r
.C("function",x=as.integer(100),y=as.integer(768))
```

R will have a copy of x and y before passing them to the C/C++ function and R will have an additional copy of the return value from the C/C++ function.

**BridgeR** middleware [6] is developed by Professor Nagiza F. Samatova’s research team (Paul Briemyer and Kora Guruprasad) using C/C++. **BridgeR** alleviates the overhead introduced by R in calling external C/C++ functions by sharing R’s and external C/C++ function’s memory space. **BridgeR** defines six data types using C++ templates, which enable bidirectional translation of objects between R and external C/C++ functions. **BridgeR** data types can create data elements in R’s memory space which can be accessed in both the C/C++ environment and in the R environment. Appendix C summarizes the design of the BridgeR middleware.

**BridgeR with stand-alone MySQL server**

The **BridgeRMySQL** package, which is newly developed in this thesis, exposes MySQL client functions as external C/C++ functions to R through BridgeR. The current BridgeRMySQL scenario calls MySQL client functions described in BridgeRMySQL package from R using BridgeR middleware. The MySQL client communicates with a stand-alone MySQL server over WAN using ODBC for database connectivity as illustrated in the Figure 2.2. In contrast to BridgeRMySQL, the existing RMySQL scenario calls MySQL client functions described in RMySQL package from R using R’s native interface to call external C/C++ functions. Since BridgeR reduces the overhead introduced by R in calling external C/C++ functions, it is envisioned that BridgeRMySQL will improve the performance of RMySQL.
BridgeR data types type cast the data from/to C/C++ data types from/to R. But the data types supported in R and MySQL are still not quite compatible to one another. So there is a need to type cast data from MySQL data types to C/C++ data types without loss. BridgeR takes care of typcasting data from/to C/C++ to/from R.

2.2.3 BridgeRMySQLD Scenario

In both the RMySQL and BridgeRMySQL scenarios, R accesses the database through TCP/IP stack. BridgeRMySQLD scenario, illustrated in Figure 2.3, takes the DB-in-Analytics approach, defined in Section 1.1, to remove the overhead introduced by TCP/IP stack during MySQL client-server communication. With the current BridgeRMySQLD approach, the MySQL server is embedded into the R-client space, so that R statistical functions can easily access the data from MySQL server through inter-process communication, thus, overcoming the overhead introduced by TCP/IP stack communication. BridgeRMySQLD scenario can be achieved with the help of embedded MySQL server, which provides all the functionalities of a database server, but still runs as a separate process in the R-client’s space and, thus, shares the R’s memory space. BridgeRMySQLD scenario embeds the MySQL server inside the R-client space as a C/C++ shared library. The above solution seems to be achievable, since R is written in C and MySQL is a mixture of C/C++.
**BridgeR with embedded database server**

With the *BridgeRMySQLD* approach, *R* will call *MySQL* client, which is an external *C/C++* function through *BridgeR*. The *MySQL* client will communicate with the embedded *MySQL* server through inter-process communication to fetch data by issuing SQL queries. The data fetched from the embedded *MySQL* server will be casted and returned to *R* as *R* native objects.

**Performance improvement through embedded MySQL**

With the *BridgeRMySQLD* approach, both the client and the server are running in the same memory space. Hence, they can communicate data back-and-forth using inter-process communication (IPC). Since both the client and the server share the same memory space, a pointer to data from the server to the client or vice-versa can be passed. *BridgeRMySQLD* entirely overcomes the overhead introduced by *TCP/IP* stack, which communicates data as data packets.
2.2.4 BridgeRMySQLDMap Scenario

In the previous BridgeRMySQLD scenario R Data Frames are used to store the data brought from the MySQL server. R Data Frames are inefficient in handling large data sets. R Data Frames repeatedly fetches the small subset of the larger data set to process them. R Data Frames also might lose data due to incompatible data types between R and MySQL.

Hence the current BridgeRMySQLDMap scenario maps the R data types into the embedded MySQL storage engine. In the BridgeRMySQLDMap scenario, whenever a Vector, List, or Matrix, or any other data structure is declared at the R-client, R memory allocation functions are overriden by allocating the memory for the data structures by the MySQL internal storage engine APIs. Allocating the memory for the data structures by the MySQL storage engine is achieved by mapping the R data structures to MySQL tables.

MySQL database server can be broadly divided into three modules: parser, optimizer, and storage engine. MySQL parser is responsible for parsing the SQL query to verify its syntactical correctness. MySQL optimizer is responsible for coming out with the optimized SQL query execution plan. Storage engines such as MyISAM, InnoDB, BerkeleyDB are responsible for storing the data as bytes.

The BridgeRMySQLD approach issues the SQL query to bring the data from the MySQL database server. MySQL database parser will parse the SQL query and then the SQL query is optimized by the MySQL optimizer. The optimized SQL query is executed, and the results will be returned to the MySQL client. During the SQL query execution, MySQL server will access the data from the storage engines through MySQL internal storage engine API. The storage engine will return data as bytes. MySQL server will convert the bytes into appropriate MySQL data type values before sending the data to the MySQL client.

Hence in the BridgeRMySQLDMap scenario, whenever R client allocate, access or modify the R data structures MySQL internal storage engine APIs are used to allocate, access or modify data by converting the data from bytes to appropriate MySQL data type values before performing required operations upon data values. Hence the BridgeRMySQLDMap
scenario effectively skips the overhead introduced by the MySQL Query parser and the MySQL Query Optimizer by directly accessing the data through MySQL internal storage engine APIs. BridgeRMySQLDMap scenario also removes the data handling by R Data Frames and efficiently process the large data sets through MySQL internal storage engine APIs.

**BridgeR with embedded storage engine**

With the BridgeRMySQLDMap approach, R will call MySQL client, which is an external function written in C/C++, through BridgeR. The MySQL client will communicate with the embedded MySQL server through inter-process communication to fetch data by skipping MySQL parser and optimizer, by directly using MySQL internal storage engine API’s, as illustrated in Figure 2.4. The data fetched from the embedded MySQL server will be casted and returned to R as R native objects.

### 2.3 Major Contributions

1. Designed the BridgeRMySQL, BridgeRMySQLD, and BridgeRMySQLDMap packages.
2. Integrated R/BridgeR with the MySQL client and the MySQL embedded server.

3. Added new MySQL client API's to the existing set of MySQL client API's by making MySQL source code modifications. The new client API accesses the data through MySQL internal storage engine API's directly skipping the MySQL parser and the MySQL optimizer.

4. Showed the proof-of-concept by implementing RMySQL, BridgeRMySQL, BridgeRMySQLD, BridgeRMySQLDMap scenarios.

5. Tested and comparatively analyzed RMySQL, BridgeRMySQL, BridgeRMySQLD, and BridgeRMySQLDMap scenarios.
Chapter 3

Design and Architecture

This chapter describes the high-level design and architecture of the system. Section 3.2 summarizes the design of the state-of-the-art RMySQL Scenario. Later sections of this chapter describe the design decisions made for the newly proposed BridgeRMySQL, BridgeRMySQLD, and BridgeRMySQLDMap scenarios.

3.1 High-level architecture of the system

Figure 3.1 illustrates the high-level architecture of the RMySQL and BridgeRMySQL scenarios that involve the stand-alone MySQL server. The R environment is made up of R Initializer and R Engine. R Engine is made up of core R statistical functions. R is extended by the R DB Interface and the BridgeRMySQL packages. The R DB Interface package, which is developed using R, is composed of the DBI package and RMySQL package. The DBI package defines the base virtual client functions to access any database, such as MySQL, Oracle, etc. RMySQL extends the base virtual client functions defined in the DBI package and defines MySQL client APIs to communicate with the MySQL server. BridgeRMySQL package, which is developed using C/C++, is made up of BridgeR package and MySQL C/C++ client APIs to communicate with the MySQL server.
Figure 3.1: High-level architecture of the RMySQL/BridgeRMySQL scenarios involving stand-alone MySQL server
Figure 3.2: High-level architecture of the BridgeRMySQLD/BridgeRMySQLDMap scenarios involving Embedded MySQL server

Figure 3.2 illustrates the high-level architecture of the BridgeRMySQLD and BridgeRMySQLDMap scenarios that involve an embedded MySQL server. R is extended with the two new packages, namely BridgeRMySQLD and BridgeRMySQLDMap. BridgeRMySQLD package, which is developed using C/C++, is made up of the BridgeR package and the Embedded MySQL server shared library. Similarly, BridgeRMySQLDMap package, which is developed using C/C++, is made up of the BridgeR package, Embedded MySQL server shared library, and newly defined C/C++ MySQL client functions.

MySQL architecture has provided the encapsulation for SQL interface, query parsing, query optimization, query execution, query caching, buffering, and pluggable storage engine. At the top of the MySQL server, as illustrated in Figure 3.1, there are the database connectors that provide access to the client applications. To the left of the MySQL server, as illustrated in Figure 3.1, ancillary tools are listed and grouped by administration services and enter-
prise services. The next layer down in the MySQL server, as illustrated in Figure 3.1, from the connectors is the connection pool layer. The connection pool layer handles all of the user access, thread processing, memory, and process cache needs of the client connection. Below the connection pool layer is the heart of the database system. At the heart of the database system, query is parsed and optimized, and file access is managed. In the next layer down below the heart of the database system is located the pluggable storage engine layer. The pluggable storage engine is one of the hallmarks of the MySQL architecture. The pluggable storage engine permits the system to be built to handle a wide range of diverse data or file storage and retrieval mechanisms. No other database today provides the facility of supporting diverse data storage mechanisms. The lowest layer of the system is the file access layer. It is at the file access layer that the storage mechanisms read and write data, and the system reads and writes log and event information. The file access layer is closer to the operating system functionalities along with the thread, process, and memory management.

3.2 Design of the RMySQL Scenario

RMySQL scenario, as illustrated in Figure 3.3, is the state-of-the-art scenario that describes how R currently communicates with MySQL through RMySQL package APIs. The R Initializer will initialize and start the R Engine. R invokes MySQL client APIs defined in RMySQL package. R will make a copy of all the input parameters sent to the MySQL client API defined in the RMySQL package during API invocation and also when API returns. RMySQL API, when invoked, makes use of application layer MySQL Client Protocol to communicate with the stand-alone MySQL server. MySQL Client Protocol will divide the MySQL client’s request into Command Packets and send them to the MySQL server through TCP/IP stack.

The stand-alone MySQL server is initialized and started by the MySQL server initializer module. MySQL Server Initializer starts the Connection Manager, which listens to the incoming new client request and dispatches the request to the Thread Manager. The Thread Manager is responsible for keeping track of threads and for making sure a thread is allocated to handle the connection from a client. The Thread Manager invokes the User Authentication Module for authenticating the user for the first time and initializes
the structures and variables containing the information on user-level privileges. The THD class represents each new thread spawned by the Thread Manager. The objects of the THD type are Thread Descriptors. Most of the server APIs are defined using Thread Descriptors as the parameter.

User is authenticated before communicating with the Connection Manager only for the first time to establish a new connection. Once the connection is established the Connection Thread will take care of processing client requests on the established connection. User Module after authentication dispatches the request to the Command Dispatcher, which invokes further lower-level modules to handle the request.

When the Parser receives the SQL query from the Command Dispatcher it will parse the query and generate a parse tree. If SQL Query is a SELECT statement, then the Optimizer will optimize the query to generate the best Query Execution Plan to answer the query. If SQL Query is an UPDATE statement then the Table Modification module will be invoked to handle the request. Similarly, depending upon the type of the SQL request, appropriate modules are invoked to generate the Query Execution Plan.

Before the query execution, Access Control module verifies that the user-client has sufficient privileges to perform the requested operation. Query execution makes use of Abstract Storage Engine Interface to communicate with any of the storage engines, such as MyISAM, InnoDB, and BerkeleyDB. Abstract Storage Engine Interface defines an abstract class named handler and handlerton structure. Each storage engine, such as MyISAM will extend and implement the functions defined in the Abstract Storage Engine Interface. The storage engine implementations will define all the operations in terms of the low-level calls of the specific storage engine.

At the MySQL server end, command packets are sent up to the application layer MySQL Server Protocol through TCP/IP stack. MySQL Server protocol will assemble the command packets together to form the MySQL client request. The assembled MySQL client request will be sent to Connection Manager to handle the request. The results of the query will be sent to the client through MySQL Server Protocol. MySQL Server Protocol will divide the result set into Result set packets and will use TCP/IP stack to communicate
these packets to the client. MySQL Client Protocol will assemble the Result set packets to form the Result set. The assembled Result Set will be given to the client.

The Query Cache module will cache the results of the current query in the buffer. When the user issues a new request, the new request will be compared, character-by-character, with the previous query and if it is the same, the results present in the query cache will be returned, otherwise, the query is processed as a new query.

The issues with the RMySQL scenario are the following:

- **R** native interface to call external functions introduces an additional overhead of copying and casting parameters to call any external R/C/C++ functions. R uses its native interface to call functions defined in any package, such as DBI, which extends R’s database access functionalities. Similarly, in the current RMySQL scenario, R uses its native interface to call APIs of RMySQL package. Hence, R introduces additional overhead both before and after calling RMySQL package APIs by copying and casting the values of the input parameters.

- TCP/IP stack involved between MySQL client-server communication introduces the overhead by communicating the request/result-set as packets.

- **R Data Frames** are very inefficient in handling large result sets from MySQL server at once. Hence, the R Data Frames will repeatedly fetch the data in small quantities and process them.

All of the above issues are addressed by the newly proposed BridgeRMySQL, BridgeRMySQLD, and BridgeRMySQLDMap scenarios, described next.

### 3.3 Design of the BridgeRMySQL scenario

The newly proposed BridgeRMySQL scenario is similar to the RMySQL scenario with the following few exceptions. As already explained in the previous section, with the RMySQL scenario, R will invoke MySQL Client APIs of RMySQL as external R functions using the R native interface to call external functions. With the RMySQL scenario, R native interface introduces additional overhead by copying and casting function parameters both
Figure 3.3: RMySQL Design and Architecture
before and after calling the APIs defined in the \textit{RMySQL} package. The overhead introduced by \textit{R} native interface to call any external functions is overcome by the \textit{BridgeR} package \cite{6}. The \textit{BridgeR} shares \textit{R}'s and any external \textit{C}/\textit{C}++ function’s memory space by providing bi-directional translation of objects between \textit{R} and any external \textit{C}/\textit{C}++ function. If \textit{R} uses \textit{BridgeR} API instead of \textit{R}'s native interface to call any external \textit{C}/\textit{C}++ function, then \textit{R} can send pointers to the variables in the shared memory space, as parameters to external \textit{C}/\textit{C}++ functions. The external \textit{C}/\textit{C}++ functions can also access and modify the values of the variables in the shared memory space with the help of pointers, which are sent as parameters. Hence, there is no need for copying and casting the parameter variables, while calling any external \textit{C}/\textit{C}++ functions by \textit{R}. Appendix \textit{C} describes the design of the \textit{BridgeR} package.

The \textit{BridgeRMySQL} package defines \textit{MySQL} client functions using \textit{C}/\textit{C}++ as opposed to the \textit{RMySQL} package, which defines \textit{MySQL} client functions using \textit{R}. The newly defined \textit{MySQL} client functions of \textit{BridgeRMySQL} package use \textit{BridgeR} as a wrapper. Hence, \textit{R} can invoke \textit{MySQL} client functions defined in the \textit{BridgeRMySQL} package through the \textit{BridgeR}, as illustrated in Figure \textit{3.4} by passing pointers to the variables declared in \textit{R}'s memory space as parameters to \textit{MySQL} client functions of \textit{BridgeRMySQL}.

\section*{3.4 Design of the BridgeRMySQLD scenario}

The \textit{BridgeRMySQLD} scenario is similar to the \textit{BridgeRMySQL} scenario with the following exceptions. As previously noted, the \textit{MySQL} client divides the request into the command packets and communicates them to the server through the \textit{MySQL} Application Layer protocol on top of the \textit{TCP/IP} stack. The \textit{MySQL} server receives the command packets and assembles them to get back the client request. The \textit{MySQL} server executes the client’s request and divides the result set into the result set packets. The result set packets are communicated back to the client through the \textit{MySQL} Application Layer protocol on top of the \textit{TCP/IP} stack. At the client end, the result set packets are assembled to get the result set back. The division and re-assembling of the data/command packets by communicating it through the \textit{TCP/IP} stack has been the major bottleneck in getting the desired performance for statistical analysis of data by \textit{R}.
Figure 3.4: BridgeRMySQL Architecture
The newly proposed BridgeRMySQLD scenario overcomes the overhead introduced by the division/re-assembling of data packets through the TCP/IP stack by embedding the MySQL Server into R. Now the embedded MySQL Server is part of R. Hence, the embedded MySQL Server runs as a separate thread in the R’s memory space, as illustrated in Figure 3.5. Therefore, R can communicate with the embedded MySQL server through the Inter-Process Communication (IPC) to communicate the command/result-set instead of using data packets through TCP/IP stack.

With the BridgeRMySQLD scenario, as illustrated in Figure 3.5, the embedded MySQL server shared library, the BridgeR package along with the C/C++ MySQL client functions are compiled together to form the BridgeRMySQLD package. The BridgeRMySQLD scenario can skip the user authentication, as the embedded MySQL server is dedicated only to the single R user. Hence, the BridgeRMySQLD scenario can send the request directly to the Command Dispatcher module by skipping the User authentication module. Therefore, the User authentication module is skipped immediately after spawning the thread for handling the incoming request by the Thread Manager. For security reasons, User authentication can still be enabled by the appropriate configuration options during the creation of the embedded MySQL server shared library.

### 3.5 Design of the BridgeRMySQLDMap Scenario

The BridgeRMySQLDMap scenario is similar to the BridgeRMySQLD scenario with the following exceptions.

The abstract class handler of the Abstract Storage Engine Module, provides methods for the basic operations, such as opening and closing a table, sequentially scanning through the rows, retrieving rows based on the value of the primary key, inserting a row, and deleting a row. Every storage engine will implement a subclass of the handler, implementing the interface methods to translate the handler operations into the low-level storage/retrieval API calls of that particular storage engine. MyISAM is one such storage engines, which has implemented all the methods of this handler class.
Figure 3.5: BridgeRMySQLD Architecture
APIs will use the handler class methods to access the data from the storage engine.

The section 2.2.4 describe the need of the new MySQL client APIs to overcome the limitations of the R Data Frames. In the BridgeRMySQLDMap scenario the MySQL client request can be in the form a simple C/C++ structure, having the information about the table name, column name and the database name as opposed to an SQL query. This thesis, in the BridgeRMySQLDMap scenario, defines the new MySQL client functions to handle the MySQL client request as a simple C/C++ structure.

The definition of the new BridgeRMySQLDMap’s client functions has been added on top of the current MySQL internal storage engine API’s, as illustrated in Figure 3.6. The difference between the new MySQL client API’s and existing MySQL client API’s is that the former will skip MySQL Query Parser, Query Optimizer, Command Dispatcher, and Access control manager, as the request is not an SQL query and uses the MySQL storage engine API’s to access data from the disk, as illustrated in Figure 3.6. Hence, the BridgeRMySQLDMap scenario can directly access BridgeRMySQLDMap Stub, as soon as a new thread is spawned by the Thread Manager to handle the incoming request.

Need of data conversion Currently, the TABLE structure represents a descriptor for a database table and the FIELD abstract class defines a descriptor for a database table field. Subclasses are defined for each column data type, such as date, varchar, double, int, etc. by extending and defining the FIELD abstract class. The handler class methods will return data as bytes to MySQL internal storage engine API’s. So BridgeRMySQLDMap API’s need to use the FIELD class to convert these bytes into appropriate values depending on the column datatypes.

The broader goal of the BridgeRMySQLDMap is to map the R’s datatypes such as Vector, List, Data Frames, etc., into the corresponding MySQL database tables/data structures for handling the data. Whenever a new R datatype is allocated/manipulated at R, the corresponding MySQL tables/data-structures need to be allocated/manipulated by the BridgeRMySQLDMap by overriding the R’s methods for handling the R’s data structures. Hence, the BridgeRMySQLDMap defines the new API’s by overriding the R’s methods for handling R’s data structures as illustrated in Figure 3.7. The following example describes
Figure 3.6: BridgeRMySQLDMap Architecture
how the $R$ Vector has been mapped as a *MySQL* table.

The following example will make some reasonable assumptions. All $R$ Vectors are defined under the database titled *VECTOR*. Each $R$ Vector is represented by a single *MySQL* table with the *MySQL* table’s name that is the same as the $R$’s Vector name. The values of the $R$’s Vector will be stored as a single *MySQL* table row. The datatype of the *MySQL* table column will be the same as the $R$ Vector’s datatype. When a new $R$ Vector is declared at the $R$’s space, a corresponding database table with the the same name as the $R$’s vector name, having a single column with the datatype as $R$’s vector datatype, under the database *VECTOR* is created. The *BridgeRMySQLD_MAP* package has defined a struct **RVector**, as illustrated in Figure 3.7 which carries all the information, such as databasename, tablename, columnname, and column datatype values, representing an $R$ Vector in the database. *BridgeRMySQLD_MAP*’s *MySQL* client function will send the request as a populated **RVector** to the embedded *MySQL* server. The *BridgeRMySQLD_MAP* package is responsible for casting the values of the corresponding *MySQL* table/datastructure to the values of an $R$ Vector.
Figure 3.7: Design of BridgeRMySQLDMap Package
Chapter 4

Evaluation

This chapter describes the configuration of the test environment, details about the various tests conducted and the results of those tests. This chapter also comparatively analyzes the results to draw conclusions.

4.1 Cost evaluation parameters for the In-database analytics with R

Cost evaluation parameters describes the parameters to be measured for evaluating a RMySQL/BridgeRMySQL/BridgeRMySQLD/BridgeRMySQLDMap scenario:

• Time to start the MySQL server
• Time to start a new client/server session
• Time to issue a query by the client and to be executed by the server
• Time to fetch the result set to the client from the server
• Time to type cast the data to the required datatypes at the client
• Time to convert the data to the required datatypes inside the server
• Time to close the client/server session
• Time to stop the MySQL server
Once the connection to MySQL server is established, the request will be issued repeatedly to fetch the data in different magnitudes. When the data with the maximum magnitude is fetched the existing connection to the MySQL server is closed. When a request to fetch data of any magnitude is made all the above evaluation parameters are measured across the APIs, which execute the request.

4.2 Test Results

The tables 4.2, 4.2, 4.2, and 4.2 tabulates the values of the evaluation parameters measured for the different scenarios when the size of the data is varied exponentially. The Figures 4.1, 4.2, 4.4, and 4.5 shows the comparative analysis of the time to query server, fetch data, typecast data, and access data by all the different scenarios. All the measurements are done at the R end using `proc.time()` procedure and are expressed in terms of seconds. Measurements are done end-to-end around a single API rather than module-to-module.

<table>
<thead>
<tr>
<th>Table 4.1: Table Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to Query Server</td>
</tr>
<tr>
<td>Time to Fetch Data</td>
</tr>
<tr>
<td>Total time for Data Access</td>
</tr>
<tr>
<td>Time to Typecast Data</td>
</tr>
<tr>
<td>Total Time</td>
</tr>
<tr>
<td>Performance Improvement from scenario 1 to scenario 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.2: RMySQL Performance Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.of elems</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10^2</td>
</tr>
<tr>
<td>10^3</td>
</tr>
<tr>
<td>10^4</td>
</tr>
<tr>
<td>10^5</td>
</tr>
<tr>
<td>10^6</td>
</tr>
<tr>
<td>10^7</td>
</tr>
<tr>
<td>10^8</td>
</tr>
</tbody>
</table>
Table 4.3: BridgeRMySQL Performance Table

<table>
<thead>
<tr>
<th>No.of elems</th>
<th>Query Server</th>
<th>Fetch Data</th>
<th>Data Access</th>
<th>Typecast Data</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QS</td>
<td>FD</td>
<td>DA=QS+FD</td>
<td>TD</td>
<td>QS+FD+TD</td>
</tr>
<tr>
<td>10</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^2</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^3</td>
<td>0.002</td>
<td>0.000</td>
<td>0.002</td>
<td>0.001</td>
<td>0.003</td>
</tr>
<tr>
<td>10^4</td>
<td>0.001</td>
<td>0.010</td>
<td>0.011</td>
<td>0.011</td>
<td>0.022</td>
</tr>
<tr>
<td>10^5</td>
<td>0.002</td>
<td>0.115</td>
<td>0.117</td>
<td>0.111</td>
<td>0.228</td>
</tr>
<tr>
<td>10^6</td>
<td>0.001</td>
<td>1.163</td>
<td>1.164</td>
<td>1.156</td>
<td>2.320</td>
</tr>
<tr>
<td>10^7</td>
<td>0.002</td>
<td>11.619</td>
<td>11.621</td>
<td>11.136</td>
<td>22.757</td>
</tr>
<tr>
<td>10^8</td>
<td>0.001</td>
<td>116.48</td>
<td>116.481</td>
<td>110.610</td>
<td>227.091</td>
</tr>
</tbody>
</table>

Table 4.4: BridgeRMySQLD Performance Table

<table>
<thead>
<tr>
<th>No.of elems</th>
<th>Query Server</th>
<th>Fetch Data</th>
<th>Data Access</th>
<th>Typecast Data</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QS</td>
<td>FD</td>
<td>DA=QS+FD</td>
<td>TD</td>
<td>QS+FD+TD</td>
</tr>
<tr>
<td>10</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^2</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^3</td>
<td>0.001</td>
<td>0.000</td>
<td>0.001</td>
<td>0.001</td>
<td>0.002</td>
</tr>
<tr>
<td>10^4</td>
<td>0.012</td>
<td>0.000</td>
<td>0.012</td>
<td>0.011</td>
<td>0.023</td>
</tr>
<tr>
<td>10^5</td>
<td>0.121</td>
<td>0.000</td>
<td>0.121</td>
<td>0.112</td>
<td>0.233</td>
</tr>
<tr>
<td>10^6</td>
<td>1.208</td>
<td>0.000</td>
<td>1.120</td>
<td>1.162</td>
<td>2.282</td>
</tr>
<tr>
<td>10^7</td>
<td>12.082</td>
<td>0.000</td>
<td>12.082</td>
<td>11.771</td>
<td>23.853</td>
</tr>
<tr>
<td>10^8</td>
<td>120.65</td>
<td>0.000</td>
<td>120.65</td>
<td>117.44</td>
<td>238.090</td>
</tr>
</tbody>
</table>

Table 4.5: BridgeRMySQLDMap Performance Table

<table>
<thead>
<tr>
<th>No.of elems</th>
<th>Query Server</th>
<th>Fetch Data</th>
<th>Data Access</th>
<th>Typecast Data</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QS</td>
<td>FD</td>
<td>DA=QS+FD</td>
<td>TD</td>
<td>QS+FD+TD</td>
</tr>
<tr>
<td>10</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^2</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^3</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^4</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10^5</td>
<td>0.050</td>
<td>0.000</td>
<td>0.050</td>
<td>0.000</td>
<td>0.050</td>
</tr>
<tr>
<td>10^6</td>
<td>0.430</td>
<td>0.000</td>
<td>0.430</td>
<td>0.000</td>
<td>0.430</td>
</tr>
<tr>
<td>10^7</td>
<td>4.270</td>
<td>0.000</td>
<td>4.270</td>
<td>0.000</td>
<td>4.270</td>
</tr>
<tr>
<td>10^8</td>
<td>42.580</td>
<td>0.000</td>
<td>42.580</td>
<td>0.000</td>
<td>42.580</td>
</tr>
</tbody>
</table>
Figure 4.1: Comparison of time to query server in secs

Figure 4.2: Comparison of time to fetch data in secs

Figure 4.3: Graph Legend
Figure 4.4: Comparison of time to access data in secs

Figure 4.5: Comparison of time to typecast data in secs

Figure 4.6: Graph Legend
4.3 Comparative analysis of the scenarios

R uses its native interface to call external C/C++ functions, it consumes lot of resources \[2.2.2\] by casting and copying the values both before and after calling the function. \textit{BridgeR} \[2.2.2\] reduces the overhead introduced to native R interface to call external C/C++ functions by sharing R’s and C/C++ memory space. Effectively \textit{BridgeR} reduces the time taken to query the database server by a factor of 6 and the time taken to fetch the data reduces by a factor of 7.5(650\%) from \textit{RMySQL} to \textit{BridgeRMySQL} scenario. Same performance improvement 7.5(650\%) is obtained to access data from the database server even in \textit{BridgeRMySQLD} scenario and \textit{BridgeRMySQLDMap} scenario compared to \textit{RMySQL} scenario due to the usage of \textit{BridgeR}. For design and architecture details of the \textit{BridgeR} see \[6\].

The following two interesting observations can be made by comparing \textit{BridgeRMySQL} and \textit{BridgeRMySQLD} scenarios. First, it seems that there is an exchange of resource consumption for querying the server and fetching the data from the database server. Secondly there is an exchange of resource consumption, the total time to access the data remains almost the same. To be more accurate the \textit{BridgeRMySQLD} consumes little more resources than \textit{BridgeRMySQL}.

\textit{BridgeRMySQL} uses TCP/IP stack to transfer data, hence accessing data takes considerable time to query data and time to transfer data. Where as \textit{BridgeRMySQLD} uses \textit{Interprocess Communication} (IPC) to transfer data, hence querying the data also brings the data to the client through IPC, which is achieved through a single API call. Since \textit{BridgeRMySQLD} does not use TCP/IP stack time to fetch data is zero.

In \textit{BridgeRMySQLD} (also in \textit{BridgeRMySQLDMap}) embedded version of MySQL is dynamically linked against R as shared library. Shared libraries consumes less space compared to static libraries, and takes additional CPU cycles(6-7) to load and execute API’s in it. This thesis speculates that this might be the cause of the increased total time utilization in \textit{BridgeRMySQLD} as against \textit{BridgeRMySQL}.

\textit{BridgeRMySQLDMap} uses MySQL internal storage engine API’s to access data by skipping MySQL Parser and Optimizer. Hence \textit{BridgeRMySQLDMap} shows performance im-
Figure 4.7: Comparing Stand-alone and Embedded server

provement by another factor of 2.83 (183%) compared as against BridgeRMySQLD (or BridgeRMySQL) and a total factor of 20.441 (1944%) performance gain compared with RMySQL.

RMySQL fetches data and temporarily stores data in R Data frames and the data will be in character format. Hence data in R data frames needs typecasting to appropriate R datatype. Currently BridgeRMySQL, and BridgeRMySQLD is not using MySQL prepared statements to query and access data, hence spending time typecasting data to the required datatype. But time to typecast data can be completely avoided by using MySQL prepared statements and BridgeR datatypes described in the C. BridgeRMySQLDMap does not need typecast data since it can be casted in the required datatype during the data conversion process. Effectively BridgeRMySQLDMap, BridgeRMySQL, BridgeRMySQLD does not take time to even typecasting the data.

4.4 Test Environment Setup

Configuration of the test environment is given in the section 4.5. The data directory ./var contains a database titled VECTOR, with a table named VECTOR_DOUBLE. VECTOR_DOUBLE table has a single column, which has more
than $10^9$ rows of numeric elements. MySQL stand-alone and embedded server can be started with the above database setup. MySQL-4.1.22 stand-alone server is listening for the new connections on the localhost at the port number 3307. MySQL-4.1.22 embedded-server is compiled with R and MySQL client programs and it can be started anytime. Accessing the same data directory at the same time by both the stand-alone and the embedded MySQL server might result into unpredictable behaviour.

### 4.5 Test Environment Configuration

The configuration of the test environment as follows:

1. Red Hat 4.1.2-42 Linux Operating System
2. GCC version 4.1.2 20071124
3. GNU ld version 2.17.50.0.6-6.el5 20061020
4. Red Hat Enterprise Linux Server release 5.2 (Tikanga)
5. There are 8 processors on the machine each with the following details
   - vendor_id: GenuineIntel
   - cpu family: 6
   - model: 15
   - model name: Intel(R) Xeon(R) CPU X5355@2.66GHz
   - cpu MHz: 2660.007
   - cache size: 4096 KB
   - address sizes: 38 bits physical, 48 bits virtual
Chapter 5

Conclusion and Future Work

5.1 Conclusion

R is a statistical analysis and graphical display environment. R’s core statistical analysis capabilities are extended by adding new statistical analysis packages to R. R packages face the challenge of accessing large volumes of data back-and-forth from a relational database, such as MySQL, in order to perform statistical analysis on these data. Transfer of large volumes of data from R to MySQL and vice-versa requires large amounts of network bandwidth, transfer time, and memory space. Transfer of large volumes of data between R and MySQL also gives rise to data governance and data redundancy issues.

The above issues are gaining importance as the size of the database is growing. In-database Analytics Technology provides a way to address the above issues by bringing R and MySQL closer to each other. There are a number of advantages in coupling R and MySQL, including data sharing, scalability, data Security, data fidelity, and others.

This thesis has explored the state-of-the-art in In-database Analytics Technology. The thesis has proposed BridgeRMySQL, BridgeRMySQLD, ans BridgeRMySQLDMap – the three novel ways of bringing statistical analysis capabilities of R with MySQL database management system closer by embedding MySQL database management system into the memory space of R’s analytical engine. The proposed three ways for embedded in-database analytics approach are implemented to demonstrate the proof-of-concept. They have been tested in a real-time environment and have improved the time efficiency of R’s access to the
MySQL database from 650% through 1944%.

5.2 Future Work

This section gives an overview of the assumptions that underly the design and implementation of the current work and how to potentially improve the performance of the different scenarios. This section also discusses the steps to be taken to deliver a complete In-database Analytics system for R. This section also briefly discusses the next steps to be taken to achieve Analytics-in-DB with R.

The embedded MySQL server claims to perform better than the stand-alone MySQL server. Our evaluation of BridgeRMySQLD, however, shows that the total access time to the database through embedded MySQL server is at least ten times slower compared to the stand-alone MySQL server. Future work will aim to find out the performance glitch to improve the embedded MySQL performance with R.

The MySQL source code does not create a shared embedded library even when configuration with “--enable-shared” is turned on; it only creates a static embedded library. Appendix A describes manual creation of the embedded shared library by compiling both C and C++ MySQL source code with the “-fPIC” option. R supports loading only shared libraries to call external C/C++ functions and does not support static libraries. Though shared libraries are lighter compared to the static libraries in terms of memory space, shared libraries typically take five-to-six additional CPU cycles against static libraries for making a function call. So this thesis speculates that the low performance of the embedded shared library with R may be due to the above reason. Hence, the problem is still open and future work should find where exactly the performance is getting hit to fix the issue.

Currently, BridgeRMySQLD and BridgeRMySQLLDMap need to start the embedded MySQL server as a process before accessing the data. There is a need to explore the possibility of removing the starting/stoping of the embedded MySQL server and use the functionalities supported by the internal storage engine API’s to access the data. Avoiding starting/stoping the embedded MySQL server will bring up additional issues, such as creating appropriate threads, locking, and sharing resources and many more system-level
issues in order to make the data access as smooth as possible.

MySQL embedded and stand-alone server performance can be tuned by setting appropriate parameter values. MySQL performance tuning would be helpful in case of delivering and deploying a complete system. If the performance of the embedded and the stand-alone MySQL server is tuned, the overall results will improve. So far, performance tuning has not been the primary focus, but it is required for real-time system deployment.

R supports a wide variety of datatypes and data structures including scalars, factors, tables, vectors (numerical, character, logical), matrices, dataframes, and lists. MySQL supports a variety of numeric datatypes, text datatypes, data and time datatypes, and string datatypes. The next goal for the BridgeRMySQLDMap is to support the mapping from MySQL datatypes to R datatypes and structures and vice versa. The current package supports read-only data, the next step would be to support allocation, deletion and modification of all these data elements. One way to achieve this is to develop and implement a new storage engine supporting the handling of all the above datatype mappings by extending the handler class.

Currently the time measurements are made across MySQL client API’s. A more appropriate model would measure the time and the space complexity of the individual modules of a database system. But the modules such as the Optimizer, Parser and Client-Server communication are quite intertwined; this complicates fine-grained module-level performance evaluation including both time and space requirements. Current measurements for time and space requirements are performed across client APIs and combine more than one module. Therefore, a more refined performance evaluation model should measure requirements for TCP/IP stack, inter-process communication, database query parsing, optimization, and internal storage engine access both separately and collectively.
Bibliography


[9] David A.James, Saikat DebRoy. The RMySQL Package(R Interface to the MySQL database version 0.6-0). 2007


Appendices
Appendix A

MySQL Installation

Here I would like to describe instructions to install MySQL-4.1.22. The system configuration is described in the beginning of the chapter on evaluation.

A.1 Installing MySQL 4.1.22

1. Add a login user and group for mysqld to run.

   ```
   >groupadd mysql
   >useradd -g mysql mysql
   ```

2. The official source for the source code download is available at [11]. Download the source from [12]. Perform the following steps as the MySQL user, any exceptions are indicated. Go to the directory where you need to unpack the distribution and change the location into it. Unpack the source code into the current directory.

   ```
   >tar xzvf /path/to/mysql-4.1.22.tar.gz
   ```

3. Go to the directory where the source code has been unpacked.

   ```
   >cd mysql-4.1.22
   ```

4. Configure MySQL-4.1.22 with the following options.

   ```
   >CFLAGS="-march=opteron -g -O3 -fPIC"
   CXXFLAGS="-fPIC -march=opteron -g -O3"
   ./configure --prefix=/usr/local/mysql
   --with-embedded-server --with-debug
   --enable-shared
• CFLAGS="-march=opteron -g -O3 -fPIC": CFLAG is an environment variable to set C compiler flags.
  – -fPIC: outputs position independent code, a characteristic required by shared libraries.
  – -g: enables debugging information. Remove this option to improve MySQL performance.

• CXXFLAGS="-march=opteron -g -O3 -fPIC": CXXFLAG is an environment variable to set C++ compiler flags.
  – -fPIC: outputs position independent code, a characteristic required by shared libraries.
  – -g: enables debugging information. Remove this option to improve MySQL performance.

• --prefix=/usr/local/mysql: setting the installation directory

• --with-embedded-server: build embedded server(./libmysqld)

• --with-debug: enables tracing the execution, which useful for debugging. It is useful only for development purposes otherwise it is recommended to not include this option during compilation, since it reduces the performance of MySQL to a large extent.

• --enable-shared=YES: build the shared libraries of the packages. This option can be made optional in our case and the need for it will be discussed in the next paragraph.

5. Compile the source code.

  > make

6. Install MySQL-4.1.22. You need to have root privileges to run the following command.

  • > make install

  • To set an option file, copy it from the support files.
>cp support-files/my-medium.cnf /etc/my.cnf

- Setup the database and run MySQL.
  >cd /usr/local/mysql
  >chown -R mysql
  >chgrp -R mysql
  >bin/mysql_install_db --user=mysql
  >chown -R root
  >chown -R mysql var
  >bin/mysqld_safe --user=mysql&

7. If you do not have root privileges, you can still install MySQL in your local home directory. Even if you have the necessary privileges, you may want to install different versions. You can do this by specifying different port numbers and socket files during MySQL configuration or specify them when the server starts. If you want to install a different version of mysql and you do not have root privileges, you can still install it in your local home directory and run mysql with a different port and socket. Instructions for installing a different version of MySQL (i.e. MySQL-4.1.22), where MySQL-5.0 is already running, are shown below.

- To install mysql in a different directory, change the --prefix during configuration of mysql to your local home directory.
  --prefix=/home/yourname/usr/local/mysql

You can specify a different port number (mysql default port number is 3306), socket file, log-file and pid-file during configuration to run multiple versions of mysql.

--with-tcp-port=3307
--with-unix-socket-path=/tmp/mysql4.sock
--log-error=/tmp/
--pid-file=/tmp/

Keep all other configurations the same, as described in the first step, except for modifying the prefix and adding new options indicated in this step.

- After the installation completes, specify the data directory explicitly to setup the database.
  >bin/mysql_install_db --user=mysql --datadir=./data

- Specify the configuration file while starting the server.
#my.cnf for running multiple versions of mysql

[mysqld]
datadir=/home/yourname/usr/local/mysql/data
socket=/tmp/mysql4.sock
port=3307
log-error=/home/yourname/usr/local/mysql/mysqld.log
pid-file=/home/yourname/usr/local/mysql/localmysqld.pid

[mysqld_safe]
basedir=/home/yourname/usr/local/mysql
datadir=/home/yourname/usr/local/mysql/data
socket=/tmp/mysql4.sock
port=3307
log-error=/home/yourname/usr/local/mysql/mysqld.log
pid-file=/home/yourname/usr/local/mysql/mysqld.pid

• MySQL server: start the server.
  ./mysqld_safe --defaults-file=my.cnf &

  shutdown the server.
  ./mysql -u root shutdown

• MySQL client: To start the client.
  ./mysql -u root -p password

  To connect to the client from a program you need to specify the port number
  and socket file explicitly.
  mysql_real_connect(mysql_init(NULL), "localhost","root",
  "password","dbname",3307, "/tmp/mysql4.sock",0)

A.2 Embedded MySQL shared library

Even by configuring MySQL-4.1.22 with “–enable-shared”, MySQL only creates
a static embedded library(./libmysqld/libmysqld.a), and not a shared embedded library. But
we have compiled the source code with CFLAGS and CXXFLAGS with “-fPIC” option,
which allows us to create a shared library. Go to ./libmysqld and create a shared library.

> cd libmysqld
> gcc -shared -Wl,-soname,libmysqld.so.1 -o libmysqld.so.1.0 *.o -lc
> ln -sf libmysqld.so.1.0 libmysqld.so
> ln -sf libmysqld.so.1.0 libmysqld.so.1
• -shared: produce a shared object which can then be linked with other objects to form an executable.

• -Wl: Pass a pointer to the linker. In our case the options to be passed to the linker are: -soname libmysqld.so.1. The value passed with the “-o” option is passed to gcc.

• -o: Output of the operation. In our case the name of the shared object to be output will be “libmysqld.so.1.0”

Set the path variable to make the library available during linking to your program.

> export LD_LIBRARY_PATH=/path/to/lib:$LD_LIBRARY_PATH
Appendix B

Embedded MySQL API’s

B.1 Datastructures for API’s

Some of the data structures necessary for client-server communication are defined in “mysql.h” under the directory “mysql-4.1.22/include/”, We need to declare the following global data structure variables for client-server communication.

```c
#include "mysql.h"
/*structure pointer for creating an object to access database */
MYSQL* mysql;
/*structure pointer for storing/manipulating the results of a query*/
MYSQL_RES* results;
/*structure pointer for storing/manipulating the individual rows of a result*/
MYSQL_ROW record;
```

B.2 Embedded MySQL Server API’s

B.2.1 Server configuration file

The embedded MySQL server group options are stored in the ./my.cnf file. A typical configuration file is shown below:

```ini
#my.cnf

[libmysqld_server]
datadir = ./data
language = ./english
```
debug=d:t:i:O,libmysqld_embedded.trace

[libmysqld_client]
language = ./english

- “libmysqld_server” group options:
  - “datadir=./data”: specifies the location of the data files. The value indicates that the data directory is present under the current directory.
  - “language=./english”: specifies the location of the character set and error message files. The value indicates that it is the same as the english directory under the current directory. Usually it is copied from /usr/local/mysql/share/mysql/english.
  - “debug=d:t:i:O,libmysqld_embedded.trace”: enables the creation of a log file titled “libmysqld_embedded.trace” under the current directory, which leaves the trace of MySQL server execution. You need to configure the source code –with-debug option to get a trace file of server execution. This is very helpful for debugging purposes.

- “libmysqld_client” group options:
  - “language=./english”: specifies the location of the character set and error message files. The value indicates that it is in the current directory. Usually it is copied from /usr/local/mysql/share/mysql/english.

The MySQL embedded server options described above are specified by the following variables.

static char *server_options[] = {"mysql_test", "--defaults-file=my.cnf"};
int num_elements = sizeof(server_options) / sizeof(char *);
static char *server_groups[] = {"libmysqld_server", "libmysqld_client"};

The server will read these options from the configuration file when they are passed as arguments to the following server API’s.
B.2.2  mysql_server_init()

API to initialize the embedded MySQL server.

/*initializing the server*/
mysql_server_init(num_elements, server_options, server_groups);

- num_elements: stores the number of elements in the server_options[] array
- server_options:
  - "mysql_test": is a label for this set of options and hence will be ignored by the server
  - "-defaults-file=my.cnf" option indicates the path to the configuration file defined above. The current value indicates that my.cnf is present in the current directory.
- server_group: indicates the group options to be read by the server and any other group option which is not included here will be ignored by the server even though it is present in the configuration file.

B.2.3  mysql_server_end()

This API will stop if any embedded MySQL server is running.

/*close the embedded server*/
mysql_server_end();

B.3  Embedded MySQL Client API’s

This section describes some of the Client API’s I have defined using the existing MySQL C API’s for client communication. Before invoking any of the following client API’s it is assumed that the embedded MySQL server is running, otherwise unexpected behaviour may be experienced by your application programs.

B.3.1  db_disconnect()

This API manages the closing of existing client database connections. It uses the MySQL C API method: mysql_close() to stop the running client.

void db_disconnect(MYSQL* db){....}
B.3.2  db_die()

This API uses the above API db_disconnect() to close the existing connection and displays appropriate error messages before closing the connection.

```c
static void die(MYSQL* db, char *fmt, ...){....}
```

B.3.3  db_connect()

This API uses the following steps to connect to an existing database.

1. Initialize the current application as a client and create a MYSQL object to communicate to the existing mysql embedded server.

```c
MYSQL* db = mysql_init(NULL);
```

2. Specify the location of the configuration file to read client group options. "MYSQL_OPT_USE_EMBEDDED_CONNECTION" indicates to MySQL that the embedded database connection should be used, rather than the regular database connection.

```c
mysql_options(db, MYSQL_READ_DEFAULT_GROUP,"libmysqld_client");
/*specifying that the client should use the embedded server instead of the regular server*/
mysql_options(db, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
```

3. Specify the username, password, server name and database name to connect to the database through the following API.

```c
if(!mysql_real_connect(db, NULL, NULL, NULL, dbname, 0, NULL, 0))
die(db,"mysql_real_connect failed: %s",mysql_error(db));
```
Appendix C

BridgeR Example

**BridgeR** package, developed by Professor Nagiza F. Samatova, Paul Breimyer and Kora Guruprasad using C/C++. *BridgeR* aims to connect the external compiled C/C++/Fortran functions with R scripting language by minimizing the overhead introduced by R in calling them. *BridgeR* API’s sends parameters to external functions as SEXP’s. SEXP stands for S-Expression, where S is the parent language of R. If you have any R variables or objects which are bound to a value, that value can be thought of as an SEXP or the structure it points to, which is SEXPREC. SEXP is an opaque pointer and its internal structure details are not exposed, so we can access values by the methods provided. So SEXP can hold any datatype from int, double, vector, list, etc...

*BridgeR* defines six new datatypes that enable bidirectional translation of objects between R and external objects. *BridgeR*Array, *BridgeR*Matrix, *BridgeR*Object, *BridgeR*ObjectList, *BridgeR*Parameters, and *BridgeR*Vector, corresponds to Vector, Array, Matrix, Object, List data types in R respectively. These datatypes can create data elements in R’s memory space, but can be accessed in both C/C++ and R environments. *BridgeR* contains the following two levels:

- User Interface level: It contains an API that provides users with C/C++ template functions, for each BridgeR data type, that enables developers to choose appropriate data types.

- Backend level: It contains both the data translation and R interface blocks. First modules handles data translation to/from R’s SEXP data type from/to C/C++ rep-
resentations. Because R is written in C, the underlying objects are defined in C/C++ data types and BridgeR can extract these representations from R’s SEXP data type using internal R functions. For example if a double vector is passed from R as an SEXP object, then the data translation block queries the R Interface to use R’s internal methods to extract the C representation of the vector.

The following example describe how to expose external C/C++ functions in R environment and how to call them also. BridgeR is written in C/C++ and supports external code written in C/C++. The steps to expose methods written in C/C++ are from [6]. For exposing methods written in C/C++ developers must do the following:

- Include “BridgeR.h”. This header file contains references to all the BridgeR data types and provides a single include statement for developers. Alternatively you can also include specific BridgeR data types required.

- Use BridgeR namespace in the header files to ease lookup of BridgeR data types and methods, and ensure that user developed codes are built within the same scope as BridgeR.

  using namespace BridgeR;

- Use the following signature for exposing a method.

  BridgeR SEXP function(SEXP arguments);

  “BridgeR” indicates that method should be made available to R, and it is an alias for “extern ”C””, a standard C/C++ construct to define variables and functions globally and is required by R. SEXP is both the input parameter type and return parameter type. arguments are the input parameters sent from R as an SEXP object.

- Build a “BridgeRParameter” object as follows, where “arguments” is the SEXP method argument.

  BridgeR::BridgeRParameters bridgerargs(arguments);

  This object will translate objects passed to and from R to external functions.
• Return an SEXP object from all exposed methods. Each BridgeR data type has a member function that return SEXP objects, such as “getRObj” method.

• Make modifications to the Makefile to build the external code into a shared library. For example:

```
all: touchcpp $(OBJS) $(PRDIR)/lib/libparallelR.so
    $(CXX) -shared -o $(libtarget) -lpthread -Wall $(OBJS) $(LIBS)
    cp $(PRDIR)/lib/libparallelR.so* .
    cp $(MYSQLDIR)/libmysqld/libmysqld.so* .
```

The following is an example of a simple C/C++ function exposed to R using BridgeR\[16\]

PARALLEL R is an alias to extern “C”.

PARALLEL R SEXP test_parameters (SEXP args)
{
    /* INPUT
    * a <- 100:105 # integer vector
    * b <- c(a, 999.01) # numeric vector
    * c <- list(y=as.numeric(0:9)) # 0 1 2 3 4 ..
    * d <- list(p=a, q=b, r=c, s=list(h=as.numeric(71:79)))
    */

    parallelr::PrParameters prpArguments (args);

    parallelr::PrVector<int> pvA(prpArguments(0,-1));
    parallelr::PrVector<double> pvB(prpArguments(1,-1));

    // Use directly once PrVector supports LSTSXP
    parallelr::PrVector<double> pvC(AS_NUMERIC(prpArguments(2,0,-1)));
    parallelr::PrVector<double> pvD(AS_NUMERIC(prpArguments(3,3,0,-1)));

    cout << "Printing A" << endl;
    pvA.print();
    cout << endl;

    cout << "Printing B" << endl;
    pvB.print();
    cout << endl << endl;

    cout << "Printing 4th element of c->y" << endl;
    pvC.print();
}
cout << endl << endl;

// Use reverse coercing instead.
pvD.print();
cout << endl << endl;

return parallelr::PrObject::getNullRObject();
}

Header file for the above .ccp file would be like below

#include "PrVector.h"
#include "PrMatrix.h"
#include "PrParameters.h"
#include "PrObject.h"
#include "PrObjectList.h"
#include "blas1_d.H"

using namespace parallelr;

PARALLELR SEXP test_projectlist (SEXP args);
PARALLELR SEXP test_int (SEXP args);

The corresponding R Script to call the above functions would be as follows

options(echo = FALSE)
dyn.load(paste("libpR_Tester_Cpp", .Platform$dynlib.ext, sep=""))

noquote("Testing PrParameters");
noquote("***************************");
noquote(" ");

noquote("Test 1");
a <- 100:105 # integer vector
b <- c(a, 999.01) # numeric vector
c <- list(y=as.numeric(0:9)) # 0 1 2 3 4 ..
d <- list(p=a, q=b, r=c, s=list(h=as.numeric(71:79)))
noquote("Input: a,b,c,d ");
print(a)
print(b)
print(c)
print(d)

result <- .External("test_parameters", a, b, d, d)
noquote(" ");
dyn.unload(paste("libpR_Tester_Cpp", .Platform$dynlib.ext, sep=""))
# unload
Appendix D

Implementation of the Scenarios

D.1 Implementation of the state-of-the-art RMySQL Scenario

An overview of the RMySQL and DBI packages are described in Appendix E.

1. Load the RMySQL package through the following command

   ```r
   >library(RMySQL)
   ```

2. Get the MySQL driver through the following command

   ```r
   >drv=dbDriver("MySQL")
   ```

3. Connect and authenticate to the MySQL database

   ```r
   >con=dbConnect(MySQL(),user="embed",password="embed1",dbname="VECTOR",host="localhost")
   ```

4. Fetch the result set and store it

   ```r
   >rs=dbSendQuery(con,statement="select * from VECTOR_DOUBLE")
   >data=fetch(rs,n=-1)
   >size=dim(data)-2
   ```

5. The result of the query will be in text form; convert it to a numeric data type and store it in an R Vector
>rsvector=c()
>for(i in 1:size)
>rsvector[i]=as.numeric(data[i,1])

6. Call any statistical function with the above vector as the input parameter. For example:

>fft(rsvector)

7. Close the result set and disconnect from the database

>dbClearResult(rsvector)
>dbDisconnect(con)

D.2 Implementation of the BridgeRMySQL Scenario

The C/C++ stand-alone MySQL server APIs enable access to stand-alone MySQL servers with the help of connectors, such as ODBC, as described in the online MySQL-4.1.22 Reference Manual[5]. The newly developed C/C++ stand-alone MySQL server APIs to R are exposed as external C/C++ functions through BridgeR[3]. The newly developed and exposed external C/C++ stand-alone MySQL server APIs are invoked from the R script to communicate with the stand-alone MySQL server.

D.3 Integration of Embedded MySQL Server with R

Embedding MySQL server into the R-client space can be achieved through the following steps.

1. Configure the MySQL open source code to generate the embedded MySQL server as a shared library. The instructions to configure and install MySQL-4.1.22, and to create an embedded MySQL server as a shared library, are described in Appendix A.

2. Develop new C/C++ MySQL client functions to communicate with the embedded MySQL server. The new C/C++ MySQL client functions take the database name as the input parameter. The new C/C++ MySQL client functions use the embedded C MySQL client APIs to fetch the data from the embedded database. The new MySQL client functions use the BridgeR data types to return the fetched data as RObjects,
such as an *R Vector, Matrix, List*, etc. The internal workings of the Embedded *MySQL* C APIs are described in Appendix E.

3. Expose the *MySQL* client functions created in the previous step as external C/C++ functions in *R*, as explained in Appendix C. Create a new shared library by combining the embedded *MySQL* server shared libraries with the *MySQL* client functions, which are external C/C++ functions to *R* and *BridgeR* shared libraries. The instructions to expose C/C++ functions to *R* using *BridgeR* and to create a shared library that combines the shared embedded database library, external C/C++ functions and the *BridgeR* shared library are explained in Appendix C.

4. Load the shared library created in the previous step using the appropriate loading functions at the *R* command prompt and call the external C/C++ *MySQL* client functions to communicate with the database to encapsulate the data in *RObjects*.

5. Invoke any *R* statistical functions with the *RObject* obtained from the previous step as the input parameter to perform statistical analysis on the fetched data.

### D.4 Implementation of the BridgeRMySQLD Scenario

- The *libRmysqld.so* file, a shared library to access data through the embedded *MySQL* server, is created by combining *libparallelR.so* (*BridgeR* shared library that connects *R* and C/C++ functions) and *libmysqld.so* (shared embedded *MySQL* server library). We need to load *libRmysqld.so* in the beginning of the *R* script to call any external C/C++ functions defined in the library.

```r
> dyn.load(paste("libRmysqld", .Platform$dynlib.ext, sep=""))
```

- Call the external C/C++ function “getvector_mysqld_R()” with the database vector name as the parameter, which accesses the database to fetch data from the database and returns an *R* vector object as the output, which is stored in an *R* object result in our case.

```r
> result <- .External("getvector_mysqld_R", "VECTOR_DOUBLE")
```

- Call any statistical function with the result as input data.
>fft(result)

- Unload libRmysqld.so.

> dyn.unload(paste("libRmysqld", .Platform$dynlib.ext, sep=""))

R supports Vectors, Matrices, Arrays, Dataframes, Lists and Factors. So there should be C/C++ API's to allocate the corresponding database elements in the database. This thesis addresses the allocation and manipulation of R Vectors as database vectors. So when R declares an R vector the assumption is that `newvector_mysqld_R()` will create a new table with the same name as the R vector in the VECTOR database and the table contains a column named “col” with the data type correspondingly mapped from R data types to database data types. I would like to describe how `newvector_mysqld_R()` will connect to the database and create tables, before describing the internals of `getvector_mysqld_R()`.

1. This step will establish client-server communication between the application program and the embedded database through the following API's. For more information about how these API's work refer to Appendix [3] The next step assumes that the client is connected to the VECTOR database.

```c
/*initializing the server*/
mysql_server_init(num_elements, server_options, server_groups);
/*Connecting to the database*/
mysql = db_connect("VECTOR");
```

2. The following SQL query will create a table, where `tablename` is the R vector name, and the correspondingly mapped R data type to database data type is the name of the column in the table.

```sql
query="CREATE TABLE" + tablename + "(col " + datatype +")";
if(mysql_query(testmysql,query.c_str())!=0)
    die(testmysql,"mysql_query_failed:%s[%s]",
          mysql_error(testmysql),query.c_str());
```

Now we have successfully created a database vector in the database.

3. Close the database connection and stop the server. For more information about how these API's will work please refer to Appendix [3]
The following steps illustrate how `getvector_mysqld_R()` will return an R Vector object. The steps for establishing and closing the client-server communication are the same as described previously.

1. Issue an SQL query to fetch data from the database.

   ```
   query = "SELECT * FROM " + table_name ;
   if(mysql_query(mysql,query.c_str())!=0)
       die(mysql,"mysql_query_failed:%s[%s]",
           mysql_error(mysql),query.c_str());
   ```

2. Fetch the results of the previously issued query.

   ```
   MYSQL_RES *res;
   if(!(res=mysql_store_result(mysql)))
       die(mysql,"mysql_store_result_failed:%s[%s]",
           mysql_error(mysql),query.c_str());
   ```

3. Declare an R vector using the `BridgeR` API as described below.

   ```
   /*allocate a R vector*/
   bridger::PrVector<T>prvA(mysql_num_rows(res));
   ```

4. The result obtained in the previous step will be in text format. Each and every row needs to be converted to the appropriate C/C++ data type defined during the creation of the database table and then it is appended to the R vector.

   ```
   /*find the number of elements in the vector*/
   T temppvalue;
   int counter=0;
   MYSQL_ROW row;
   while((row=mysql_fetch_row(res)))
   {
       std::stringstream ss((char*)*row);
       ss>>temppvalue;
       prvA[counter]=temppvalue;
       counter++;
   }
   ```
5. Return the \( R \) Vector as an \( R \) object using BridgeR API.

\[
\text{return prvA.getRObject();}
\]

D.5 Implementation of the BridgeRMySQLDMap Scenario

Design of the class \( \text{RMySQLDMap} \) and its methods are illustrated in the Figure 3.7 of the design and architecture chapter. This section describes how the class \( \text{RMySQLDMap} \) has been added to the MySQL open source and how it is accessed by the \( R \) script.

MySQL has an \textit{Abstract Storage Engine} class \textit{handler}, which has been extended and implemented by various \textit{Storage Engines}, such as \textit{MyISAM}, \textit{InnoDB}, etc. The \textit{Abstract Storage Engine} class \textit{handler} supports basic low-level operations such as opening a database table, closing a database table, updating/modifying a database table etc. MySQL has defined higher-level \textit{Storage Engine wrapper APIs} on top of the \textit{Abstract Storage Engine} class \textit{handler} to establish transactions with the \textit{Storage Engines}, managing database tables with the help of the \textit{TABLE} descriptor for an individual database table, etc. The methods of the \textit{RMySQLD} class are defined by the \textit{Storage Engine wrapper APIs}. Implementation of one of the methods of the \textit{RMySQLDMap} class to fetch data from the database is described below.

```c
int RMySQLDMap::R_fetchdata()
{
    /*API to leave server execution trace*/
    DBUG_ENTER("Entering R_fetchdata");
    TABLE* table;/*Table descriptor for an individual database table*/

    /*open_table is the Storage Engine Wrapper API to open a database table*/
    /*assuming that table does not has any alias parameter is made null*/
    /*find out what is the need of refresh but currently assumption is true*/
    /*column is an instance of the newly defined data structure RVector
    representing an R Vector*/
    if(table=open_table((THD*)mysql->thd, column->db_name,
        column->table_name, column->table_name,&refresh))
    {
        /*API to leave server execution trace*/
        DBUG_PRINT("info",("rows in the opened table
        \%d ",table->file->records));
    }
}```
/*Depending upon the datatype of the R Vector memory is allocated*/
if(column->column_datatype==1)
    column->datapointer=
        new longlong[table->file->records];
else if(column->column_datatype==2)
    column->datapointer=
        new double[table->file->records];
else
    column->datapointer=
        new double[table->file->records];/*default case*/

/*reading the first row of the table*/
if(!table->file->read_first_row(table->record[0],table->primary_key))
{
    if(column->column_datatype==1)
        (static_cast<int*>(column->datapointer))[0]=
            table->field[0]->val_int();
    else if(column->column_datatype==2)
        (static_cast<double*>(column->datapointer))[0]=
            table->field[0]->val_real();
    else
        (static_cast<double*>(column->datapointer))[0]=
            table->field[0]->val_real();

    /*number of elements pointed by the pointer*/
    (column->datasize)++;}

/*reading the remaining rows*/
DBG_Print("info","\nvalue fetched:\\%f",
    (static_cast<float*>(column->datapointer))[0]);
for(int i=1;(i<column->datalimit)&&(i<table->file->records);i++)
{
    if(!table->file->rnd_next(table->record[0]))
    {
        if(column->column_datatype==1)
            (static_cast<int*>(column->datapointer))[i]=
                table->field[0]->val_int();
        else if(column->column_datatype==2)
            (static_cast<double*>(column->datapointer))[i]=
                table->field[0]->val_real();
        else
            (static_cast<double*>(column->datapointer))[i]=
                table->field[0]->val_real();
        }
DBUG_PRINT("info","\nvalue fetched:\%f",
(static_cast<float*>(column->datapointer))[i]);

/*number of elements pointed by pointer*/
(column->datasize)++;
}

/*Storage Engine Wrapper API to close table*/
intern_close_table(table);

DBUG_RETURN(0);
}
Appendix E

Overview of R database packages

E.1 R/S-PLUS Database Interface (DBI)

All classes in the DBI (R/S-Plus Database Interface) package are virtual; therefore, it needs to be extended by specific R/DBMS implementations. The base classes for all DBMS connection classes are shown below:

```r
class DBICONNECTION
class DBIDRIVER
class DBIRESULTSET
```

Individual drivers (ODBC, Oracle, PostgreSQL, MySQL, etc.) extend this class in a database-specific manner. The methods and parameters inside the interface are as shown in the diagram. In the diagram, “.....” indicates that the list of these parameters can be anything depending on the class, which implements these interface.

DATA TYPE MAPPINGS  Data types supported by databases are different than the datatypes supported by R. Mapping primitive datatypes is straightforward. User-defined functions can be specified to perform the actual conversion.

OPEN ISSUES AND LIMITATIONS OF DBI PACKAGE

- Non-SQL: Is it realistic to attempt to encompass non-relational databases, like HDF5, Berkeley DB, etc.?
Figure E.1: DBI Package
• Security: allowing users to specify their passwords on R/S scripts may be unacceptable for some applications. We need to consider alternatives where users could store authentication in files with more stringent permissions. Perhaps we can address this issue by using encrypted passwords (this prevents users who do not want to know the password from viewing it, but does not prevent those who want to know by decrypting).

• Exceptions: the exception mechanism is a bit too simple, and it does not provide information when problems stem from the DBMS interface itself. For instance, under/overflow or loss of precision as we move numeric data from the DBMS to the more limited primitives in R/S.

• Asynchronous communication: most DBMS systems support both synchronous and asynchronous communications. DBI has not yet specified how to interrupt the server plus other details. Some DBMS systems use threads to implement asynchronous communication, something that neither R nor S attempts to address.

• SQL Scripts: DBI only defines how to execute one SQL statement at a time, forcing users to split SQL scripts into individual statements.

• BLOBS/CLOBS: large objects (both character and binary) present some challenges to both R and S-Plus. It is yet to be defined how to deal with binary objects in R.

• Transactions: Transaction management has not been fully described.

• Binding Variables: Iterating over a list of rows maintained by a variable when specified in an embedded SQL statement is not supported yet.

### E.2 RMySQL

RMySQL\[9\] is the package for communicating with the MySQL Database. The R client component of the database communication is initialized through MySQL(), but connecting to the database engine needs to be done through a call to dbConnect(). MySQL() provides connectivity to the backend database server through the following API.

```r
>MySQL(max.con=16, fetch.default.rec=500, force.reload=FALSE)
```
• max.conn: the maximum number of connections that are permitted to be open at one
time. Theoretically, there can be any number of connections to the server from clients.
RMySQL code uses a very simple linear search algorithm to manage its connection
table.

• fetch.default.rec: the number of records to fetch at a time from the database

• force.reload: should the client code be reloaded. Setting this to TRUE allows you
to change default settings. One important thing to remember is that all connections
should be closed before re-loading.

The R client component of the database communication is initialized, but note that con-
necting to the database engine needs to be done through calls to dbConnect().

Example of using RMySQL  Create a MySQL instance and create one connection.

> m<-dbDriver("MySQL")

Open the connection using username, password, etc., as specified in the “[authentication]”
section of the config file $HOME/.mycnf

> con<-dbConnect(m,group="iptraffic")
> rs<-dbSendQuery(con,"statement")
> df<-fetch(rs,n=50)
> dbHasCompleted(rs)
[1] True
> dbClearResult(rs)
Figure E.2: RMySQL Package