The Pennsylvania State University

The Graduate School

DEVELOPING A JAVA BINARY OBJECT TECHNIQUE FOR TRANSFERRING HIGH VOLUMES OF DATA IN DATA CENTERS

A Thesis in

Computer Science and Engineering

by

Nicholas Adam Jarvis

© 2008 Nicholas Adam Jarvis

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2008

The thesis of Nicholas Adam Jarvis was reviewed and approved* by the following:

Chita R. Das
Professor of Computer Science and Engineering
Thesis Advisor

Padma Raghavan
Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School

ABSTRACT

As companies deploy large data centers to provide data storage and access services to their customers there is a growing need for high performance solutions for data analysis. Data centers are becoming more prevalent with the increase in storage capacity, the cost reduction in high powered hardware, and the concrete web paradigm that these systems offer. The generic three-tiered approach of most data centers grants sufficient abstraction between the presentation layer, business logic layer, and data persistence layer, and provides web applications with an assortment of services. Most traditional data centers rely on HTTP protocol utilizing XML and HTML requests and responses as well as technology built around Java. Because of the increase in overhead (both from generic web protocol and Java serialization), data centers are having difficulty employing large data retrieval services for high performance analysis tools. It is well known that the greatest latency of these systems is due to network latency from the client workstations to the first tier of the data center. To alleviate this latency, some applications turn to services that empower low level communication such as TCP/IP and RMI.

Our research tries to reduce the latency of such data intensive services through basic HTTP protocols and Java solutions. By adhering to standard web approaches, such common issues as network and firewall configurations, as well as data format inconsistencies, can be avoided. This research provides an effective solution to high volume data request latency problems and provides a generic methodology that reduces implementation overhead in the business logic layer of data centers. Based on platform independent Java technology, a generic binary transfer object will be presented that

allows flexible transfers of relational data over traditional web protocols. Additionally, serialization techniques are improved, allowing more access to information stored in data centers. A comparison will be presented between the popular binary transfer approach of plain old Java objects (POJO) and the flexible binary methodology developed.

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES viii

ACKNOWLEDGEMENTS ix

Chapter 1 Introduction 1

1.1 Three Tier Data Center Paradigm 2
1.2 Research Objective 4
1.3 Organization of Thesis 6

Chapter 2 Related Work 7

2.1 Data Formats and Object Transfer Methods 7
2.3 Plain Old Java Objects and Object Relational Mappings 10
2.4 Java Serialization 13

Chapter 3 A Flexible Approach for Transferring Relational Data 17

3.1 Research Data Center Specification 17
3.2 High Volume Transfer of Relational Data 20
3.2.1 Using Java Objects for Relational Data Transfer 20
3.2.2 Mapping Java Primitive Values to Relational Data 23
3.2.3 Using Metadata and XML for Dynamic Mapping 25
3.3 Serializing Binary Data Objects 28
3.3.1 Java Externalizable Interface 29
3.3.2 Binary Data Format 30
3.3.3 Additional Techniques for Faster Transfer 33
3.3.4 The JDatabaseRecord Serialization Process 38
3.4 JFlexBOT: Java Flexible Binary Object Transfer 40
3.4.1 Developing a Relational Data Access Service with JFlexBOT 40

Chapter 4 Experiments and Results 44

4.1 System Setup 44
4.2 Experimental Procedure and Metrics Definition 45
4.3 Results and Comparison 48

Chapter 5 Conclusions and Future Work 57

5.1 Conclusions 57
5.2 Future Work 59

Bibliography 61

Appendix A JDatabaseRecord.java Source Code 66

Appendix B ColumnTypeEnum.java Source Code 77

Appendix C Java / SQL Mapping Source Code 78

Appendix D POJO Experiment Metrics 81

Appendix E JFlexBOT Experiment Metrics 83

LIST OF FIGURES

Figure 1-1: Java EE application server diagram 3

Figure 3-1: System architecture of reference data center 19

Figure 3-2: ORM example 21

Figure 3-3: JDatabaseRecord primitive mapping 24

Figure 3-4: Database metadata XML example 26

Figure 3-5: Illustration of extra bytes from contrasting data format 32

Figure 3-6: Database fields double value representations 33

Figure 3-7: JDatabaseRecord setValues() pseudo-code 35

Figure 3-8: Single Bit Algorithm (SBA) for transferring zero and null data............... 37

Figure 3-9: Pseudo-code for storing data in a JDatabaseRecord 39

Figure 3-10: Data center data access service workflow with JFlexBOT approach 42

Figure 4-1: Comparison of 25000 records 49

Figure 4-2: Comparison of 100000 records 49

Figure 4-3: Comparison of 250000 records 50

Figure 4-4: Comparison of 1000000 records 50

Figure 4-5: Average bytes per record 52

Figure 4-6: Average throughput for small, medium, and large records....................... 53

Figure 4-7: Experiment latency metrics for 1 million large records........................... 54

LIST OF TABLES

Table 3-1: Java primitive data types 23

Table 4-1: Experiment Matrix 47

ACKNOWLEDGEMENTS

I would like to first and foremost thank my wife, Jaime, for her endless support and patience during the years of my class work and research. I would also like to thank my advisor, Chita R. Das, for his guidance, input, and help throughout my research. Finally, I would like to thank my colleagues, co-workers, family, and friends for their inspiration, support, and aid for both my research and career.

Chapter 1

Introduction

The World Wide Web (WWW) has become a platform for developing and deploying Rich Internet Applications (RIA) utilizing data center technology. Today, web systems are developed using a model driven concept that separates the graphical user interface from the business logic that performs the requested actions. With this approach, applications have enabled massive amounts of information to be available to businesses, consumers, and end users through a rich internet experience. The move from the "information age" to the customer focused or "participation age" [18] is causing many system architects to redesign their web applications. With the rise of such architectures such as Java Enterprise Edition (EE), web applications are becoming easier to develop, more efficient to deploy, and more pleasing to use. As disk storage, CPU speed, and network bandwidth increase, data centers need better techniques to provide data to end users faster. Furthermore, platform independent solutions must be developed so that end users exploiting different computer systems have access to this data.

Most data centers deployed on the internet cannot efficiently provide large amounts of data to end users due to network latency and traditional server side programming shortfalls. The recent conversion of legacy enterprise applications to rich internet applications has driven user expectations, and users expect more information to be at their fingertips quickly. This is especially true for high performance scientific applications that access massive amounts of data for analysis and computation. However,

current serialization techniques do not allow data to be transferred efficiently. Our research focuses on the transfer of high volumes of relational data over traditional web protocols for analysis. Using web and data center paradigms, we aim to provide faster and more efficient binary transfers of data.

1.1 Three Tier Data Center Paradigm

Data centers are the desired system architecture of a wide array of web applications deployed on the Internet. These systems are modeled using a well structured, three-tiered approach enabling each layer to provide a level of abstraction for a highly functional system. Within this paradigm, the three separate layers perform unique tasks and can be disseminated in the following manner. The first tier consists of a web server or a cluster of web servers that forward Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol over Secure Socket Layer (HTTPS) requests to the middle tier [5, 28]. Usually the first layer consists of a few clustered servers used to load balance a multitude of requests to the middle tier. The second tier contains application servers used to process requests and perform business logic based on the type of request. The most common configurations are clustered high powered servers with large memory and multiple processors. Finally, the third tier provides data persistence usually through database servers that are connected to the middle tier and return data requested by the application servers. The third layer of these systems is mostly deployed with high performance servers running database management systems such as Oracle 10g, Sybase, or MySQL.

Java EE has become one of the most popular middleware technologies for deploying data centers on the web. Formally known as J2EE, the specification defines interfaces, APIs, methodologies, and a run time infrastructure for developing and deploying distributed internet Java server applications [1]. Java EE allows enterprise applications to be developed in a distributed server environment using a host of Java specifications. Applications are built on components, and the Java EE container controls all transactions, security, scalability, and component management. Java EE traditionally has two containers, a web container that handles all web applications through Java Servlets [26] and JSP pages, and an Enterprise Java Bean (EJB) container that handles messaging, security, and other EJB technologies.

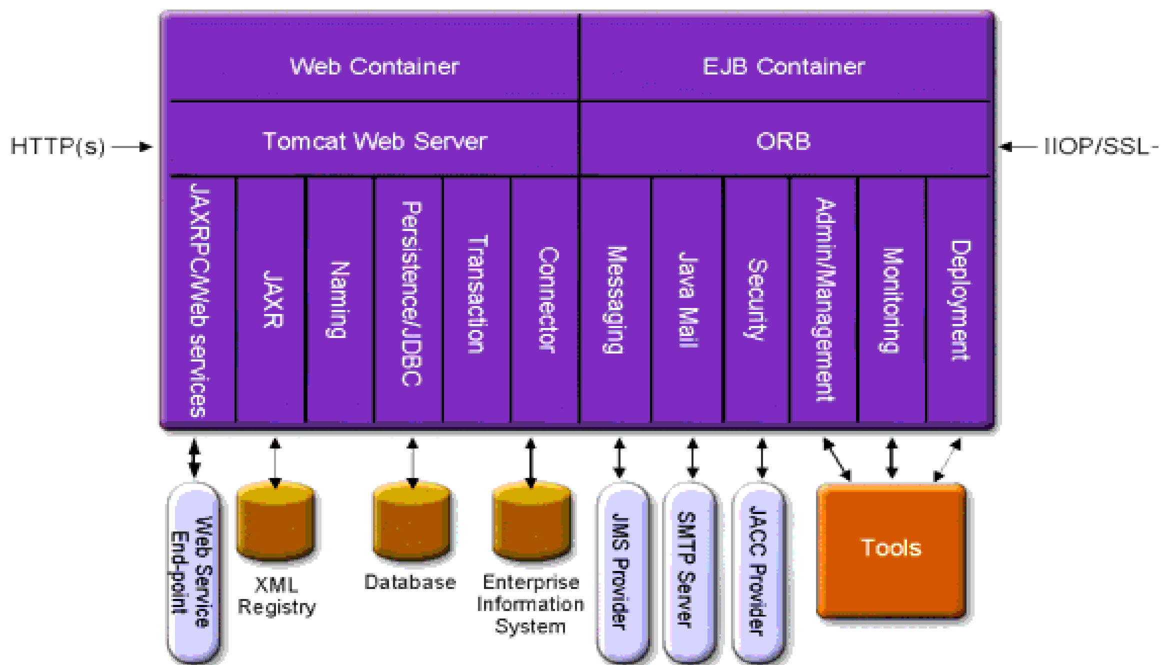


Figure 1-1: Java EE application server diagram

Figure 1-1 illustrates the architecture paradigm of the two containers utilized in Java EE [58]. The Java EE web container enables HTTP and HTTPS requests to be serviced by a Web Server as depicted in the figure. APIs and Interfaces such as Java API for XML based Remote Procedure Call (JAXRPC), Java API for XML Registries (JAXR), Java API for XML Binding (JAXB), and Java Database Connectivity (JDBC) provide robust server development frameworks utilized to build web applications. Additionally, this container allows connectivity with web services, XML, and an array of databases. The EJB container requests are traditionally processed over Internet Inter-Orb Protocol (IIOP) or Secure Socket Layer (SSL). IIOP is the low-level protocol for Common Object Request Broker Architecture (CORBA) which enables distributed applications written in different programming languages to communicate. This enables server side connectivity to other applications, as well as mail servers and Java Message Service (JMS) providers. Additionally, the EJB container supplies management, monitory, and deployment capabilities for deployed web tools. These two Java EE containers have streamlined the deployment of many web applications and have provided robust capabilities and connectivity for present web technologies.

1.2 Research Objective

The largest latency in data center requests can be traced to the WAN connection between a client machine and the first-tier of the data center. This is due to the uncertainties of the network bandwidth and the network workload of the WAN. Protocols such as HTTPS place additional communication overhead on these requests, making it

more difficult to transfer large amounts of data to a client. Many data centers develop distributed data transfer solutions over high performance networks [2, 7, 30, 50] to alleviate these problems. By focusing on high performance solutions, a large user base is prevented from accessing this data. Furthermore, network and firewall configurations can prevent access to additional end users. Because of this, there is a need to allow web applications to access this data.

Our research focuses on providing high volumes of analytical data to web applications using conventional web protocols. We develop a platform independent Java implementation that enables relational data to be serialized as binary data and transferred from a data center using HTTP or HTTPS. This approach is generic and allows dynamic mapping of relational data to Java binary objects. By providing a generic solution, we can encompass any format of data stored in flat relational database tables. Additionally, software development time is reduced when new databases are added to the data center because new Java objects do not have to be created for each new data. This solution improves performance by reducing the size of the data transferred and thus increases throughput. We alleviate the performance issues that occur from using traditional binary approaches such as plain old java objects (POJO). This is important because it enables massive amounts of data to be accessible to web applications that were previously constrained by poor performance. With our solution, data centers will be able to provide larger amounts of data quicker. Although this is not a new concept, the serialization techniques and generic mapping implementation provide a different and effective solution to these problems.

The target data center of our research incorporates known Java EE and server technologies as previously discussed. Analysis of data is accomplished using a Java web application built on such technologies as Java Swing, Java OpenGL, Web Services, and XML. Using the system, we are able to compare our flexible binary object transfer approach with the previously used POJO methodology. All experiments were conducted on the live system, allowing accurate result metrics to be obtained.

1.3 Organization of Thesis

Chapter 2 provides an overview of the related work associated to this research. The conglomeration of data formats and transfer techniques, Java serialization, and object relational mapping are used as a springboard for the research. Chapter 3 details the data center architecture as well as the implementation specifics of the binary transfer approach. Chapter 4 presents experiments and results to compare the proposed technique to a traditional data center object transfer methodology. Chapter 4 also provides a comparison and analysis of the experiments and methodologies. Finally, Chapter 5 concludes the research and offers further direction into future work.

Chapter 2

Related Work

There are three distinct areas of interest that relate to the conducted research, therefore, the related work chapter is divided into three sections. First, proposed approaches of formatting and transferring data across networks will be discussed. Next, associated work in the area of object relational mapping (ORM) and Plain Old Java Objects (POJOs) is summarized. Finally, related work in the area of Java Serialization and data marshaling are outlined. Although all sections are explicitly unique, our research correlates this related work and assembles additional research from these topics.

2.1 Data Formats and Object Transfer Methods

Data centers usually utilize the service oriented architecture (SOA) paradigm for the deployment of web services. SOAs follow the producer consumer model which enables subscribers to obtain services from a producer. With more architectures adhering to this paradigm, many services are exposed allowing efficient access to data using generic protocols. Most of the research in this area is conducted on Simple Object Access Protocol (SOAP) [11], Java Web Services (JAX-WS) [24], and much of our work is influenced from Java distributed middleware comparisons as in [12, 29, 35, 36, 38, 42, 44]. These web service technologies encode and decode data using Extensible Markup Language (XML) [4] formatted as strings. Depending on specific formatting styles, each

character of a string can be as many as 4 bytes of data. Many large data center sites, such as Google and Yahoo, have moved to a SOA model in order to offer their services in a generic and flexible format through web services. SOAP is inherently performance bound because of the format and encoding used to transmit XML over the wire. The question then becomes, what are the different formats and techniques available to decrease latency for web application services?

Google's resolution to the data format problem [14] was recently revealed to the open source community. This project, called Protocol Buffers, is Google's method of generically defining simple data structures for fast serialization transfer. To work with Protocol Buffers, a user defines message types in a record structure using key value pairs. Then a Protocol Buffer compiler is run on the message types to generate data access classes that have their own optimized serialization and de-serialization code. New data fields can be added to the format without breaking backwards-compatibility. To accomplish this, old binary classes compiled with the Protocol Buffers approach ignore new fields, so the protocol can be extended without breaking existing code [15]. The greatest benefits of this work include platform independence, language neutral conformity, simple data format flexibility, and transfer speeds of 20 to 100 times faster than XML. However, this approach is limited by its dependency on a special definition language and the Protocol Buffer compiler. Data fields removed from the structured message format will violate the protocol, so it does not provide true flexibility.

Additional work has been reported for transferring objects over high performance networks [2, 3, 7, 22, 30, 50]. A generic Java object transfer API called JToe is developed in [2]. This approach utilizes an IBM developed Java virtual machine (JikesRVM) [21],

enabling improved memory access and garbage collection. JToe implements zero copy communication over TCP and provides an API for sending arrays of data as memory addresses. A JToe data stream consists of a size specification, class header, instance header, small objects, large objects, and references. The developed low level communication object transfer shows about 50% improvement over conventional object transferring. A high performance network protocol called SABUL [30] is another data transfer protocol. This work focuses on improving the network protocol stack by exploiting TCP's control connection and UDP's data connection. SABUL divides its packets into a data packet and three feedback control packets. The research demonstrates how simple UDP data algorithms can improve the performance of data transfer. Research conducted in [7, 50] discusses a Java object transfer library called Expresso. This approach also targets high performance networks, but relies on a cluster of computers on the network which share similar memory resources. Expresso transfers Java objects using an ISO-address space by placing the transferable objects at the same virtual memory address on the receiver as on the sender. The transfer becomes a memory block transfer analogous to a DMA operation [7].

Our research focuses on dynamically transferring binary data stored in relational databases through data access services. This includes defining a data format that will allow efficient transfers of relational data to a client. Towards this regards, cited is a United States Patent for formatting tabular data to be sent across a stream from a client to a server [13]. This technique called Advanced Data Table-Gram (ADTG) format, consists of a header section, handler options section, a row section, and an end section [13]. The proposed format takes relational data from a rowSet and creates an ADTG; the ADTG

message is then wrapped in a SOAP message and sent to the client. This approach has the added benefit of the SOAP protocol in which data is sent as messages in a machine and language independent manner. It does however result in reduced performance because of the additional SOAP transfer overhead.

Our work builds on these approaches by defining a simple binary data transfer object that will encapsulate relational data and provide fast transfer times from a data center. Unlike [14], our approach will enable various data fields to be added or removed from the object to incorporate dynamic field definition. Although our methodology is modeled off of a Java binary object, it will not include the overhead associated with Java POJOs. The serialization and deserialization process of Java POJOs cannot handle dynamic changes to the data member specification. Rather, our paradigm is as flexible as XML and SOAP because we don't levy restrictions on format structure and we allow dynamic runtime manipulation for data transfers.

2.3 Plain Old Java Objects and Object Relational Mappings

Plain old Java object (POJO) is an implementation methodology presently used to store and retrieve information from databases in a three-tiered web application's business logic layer. POJOs map private member fields to database data using public getter and setter functions. The getter and setter functions are implemented to store and retrieve the persisted data values. POJOs interface between the business logic layer and the persistence layer of data centers. In the three-tiered model of most data centers deployed on the internet today, a majority of the systems store data in relational databases and

utilize a persistence layer as a level of abstraction. This methodology contrasts the object oriented approach and inherently causes design and implementation difficulties. Because of these difficulties, research has been concluded to bridge the gap between tables and objects [10, 19]. In [10], tables and documents are encompassed through object oriented languages through the addition types and expressions. Although XML and SQL have become popular, the bridge between relational, hierarchical, and object oriented data seems too difficult to overcome. The object paradigm traverses objects via their relationships, whereas the relational paradigm join rows of a table via their relationships [19, 56]. The impedance mismatch, based on combining fundamental technologies such as hierarchical XML, object oriented languages, and relational databases, has pushed software engineers to become knowledgeable in cross disciplines that help develop integrated solutions at an enterprise level.

To overcome the differences between the object oriented methodology and relational databases, there have been several open source and commercial ORM persistence solutions [31, 32, 33, 34, 43, 45, 49]. One of these solutions, called Hibernate [45] hides the persistence layer details from the business logic tier of a data center. Hibernate provides an interface between persisting data and a relational database using XML files configured from database tables and schemas. POJOs are dynamically created by Hibernate to provide developers with a domain object that can be managed between the UI and business logic layers [48]. POJOs are simple objects that permit binary serialization across the byte stream, and unlike XML which encodes data as strings, binary data transfer has improved serialization performance. Additionally, POJOs are extremely easy to use because developers only have to implement getter and setter

functions. With this approach, Hibernate eliminates the complexity of persisting data and provides a solution to the ORM problem. With Hibernate, as well as the success of other open source ORM solutions such as JPOX [28], Java has integrated Java Persistence API (JPA) into their Enterprise Edition package. JPA offers a standard persistence layer similar to other open source implementations. JPA supports integration with various industry database management systems, however JPA can persist data without amalgamating with a database layer. Furthermore, popular persistence layer implementations can be combined with JPA as a unified solution. This flexibility has allowed JPA to gain popularity with the developer community.

The ORM problem and preceding solutions provide a web layer in data centers that conceals database management system complexity from developers trying to store data. The plain old Java object approach correlating to the ORM issue provides a straightforward methodology to accessing and persisting data. We want to utilize this approach by developing a simple binary transfer object inspired by plain old Java objects. We plan to develop an uncomplicated, yet extensive binary object that is capable of mapping relational data to be transferred over the wire as efficiently as possible. Solutions such as Hibernate and JPA create a POJO for every logical object to be stored in a relational database, and this becomes tedious and cumbersome especially in a rapidly changing environment. As more tables, schema's, or even databases are added or changed, new POJO objects need to be created for each adjustment. This makes both the client and server dependent on these updates, and thus, development time is delayed and a quick turnaround of a system is no longer achievable. Our research takes these concerns

into account and improves upon them to develop a flexible binary object transfer solution.

2.4 Java Serialization

The standard Java serialization and de-serialization procedure does not perform well for high performance distributed environments. Java serialization performance is poor because redundant information is transmitted when an object is serialized. In addition, large string encodings for class package referencing and lengthy algorithms to preserve flexibility are employed. The default implementation allows any Java object to be serialized through the java.io.serializable interface declaration [52]. Simplicity is the key advantage; an object can be serialized by the simple writeObject() method and deserialized by the corresponding readOjbect() method [17]. However, the generic methodology of Java serialization results in high overhead and thus reduces performance. To alleviate Java's serialization inadequacies much work has been performed [6, 17, 20, 27, 53, 55]. Collectively, the related serialization research strives to solve the same issue; determine how minimal data can be transmitted across the stream for Java object reconstruction.

UKA-serialization is an extended version of Java Serialization that encompasses such techniques as slim encoding, internal buffering, and reflection enhancements [6]. This work is dependent on UKA-aware objects, objects in which explicit marshaling and unmarshaling routines are specified. Explicit marshaling and unmarshaling methods provide a majority of the performance improvement in their work. In addition, further

improvements are made by textually encoding class names and package prefixes into smaller representations. UKA-serialization also has full control over buffering. Buffers are declared public and can be accessed during the marshaling process. This approach enables buffer instantiations to be directed by byte size, rather than the more general approach taken in Java serialization that is geared towards TCP/IP interoperability. UKA-serialization's technique improves upon Java's serialization process significantly.

Similarly, research in [17] makes improvements to the string representations of class names and package prefixes. In this work, it is noted that redundant package names of serialized objects are causing large overhead, and their approach is to remove redundancies in the string representation of packages. To improve this deficiency, a 2-byte code is assigned to the common parts of the package structure during the initial transmission. The 2-byte code is stored in a package data structure. All subsequent serializations of the same class will result in a direct lookup into a tree, resulting in less data sent across the wire. This is accomplished using a hash table and an array to map names to encodings and encodings to names [17]. Results show that smaller object serialization sizes are attained up to 50%. However, the introduction of a time delay can be excessive for some smaller object serialization.

Serialization is improved in RMI systems [20, 27] through different binding techniques. Dynamic specialization [20] maps memory directly from a sender's representation to receivers by sending a layout description message passed during RMI binding time. Consequently, the byte sequence in the network buffer correlates with the receivers object representation and a specialized serialization routine can be formulated

from the layout descriptors dynamically. On the contrary, partial evaluation for faster RMI is a serialization technique that consists of three phases [27]:

1. Binding-time analysis (BTA); a source program and a list of binding-times that returns an annotated program in which every construct has a binding time (static or dynamic)
2. Partial evaluator is generated based on BTA
3. Dynamic compilation is performed and partial evaluator generates specialized serialization code

Both techniques exploit specialized code to efficiently serialize RMI arguments across the stream.

As previously mentioned, explicit marshaling and unmarshaling can be exploited by the java.io.Externalizable interface to improve the performance of serializing Java objects. Previous research uses these ideas as well as procedures to remove redundant data and reduce bytes transferred over the stream. Specialized serialization code and data reduction are methods we utilize in our current work also. We employ the java.io.Externalizable interface, and we build off the methodologies in [6, 17, 20, 27, 53, 55] to improve our serialization process. In addition, our observation that database data is homogenous and structured allows us to exploit methods not discussed previously. Unlike most serialization research that aims to improve serialization, while maintaining object references and tree structure, we can advance our methods because of the characteristics of relational data. This unlocks constraints bestowed on previous RMI and serialization research.

By conglomerating these ideas, we conduct research to provide a binary data transfer approach for data access services in three-tiered data centers. Our proposed work will spring board off the inherent relationship of simplified object transfer mechanisms and object relational mapping. In addition, we are able to exploit simple serialization concepts to improve Java object serialization. Altogether, in the next chapter we discuss our implementation of a Java flexible binary object transfer methodology.

Chapter 3

A Flexible Approach for Transferring Relational Data

Although data centers with simple transactions and small data footprints are still deployed presently, there is an increasing need for systems to supply significant amounts of data for use in high performance analytical web applications. As more data centers are created to store high volumes of data, traditional implementations do not suffice. Additionally, rapid changes made to these systems are accruing large developmental overhead because of the one-to-one correlation between private data members and columns in a database. Therefore, when schemas change, structures are redefined, or databases are added to the system, overhead accumulates. To alleviate these problems, we have developed a flexible binary Java object capable of dynamic data mapping. Our work is completed on an operationally deployed data center storing tera bytes of analytical data. Using this data center, we are able to compare traditional POJO methodologies to our approach.

3.1 Research Data Center Specification

The reference system used for this research is an operationally deployed data center developed, integrated, and maintained by Raytheon Systems, Inc. The data center is used by several sectors of the defense industry and provides immense storage facilities, 99% percent fail safe availability, and high volume data access services for web analysis.

Due to the sensitivity and classification level of the system, only certain details can be provided in this research. Nevertheless, the system adheres to current web data center paradigms and consists of the three-tiered architecture as described above. The first tier consists of two high powered SunFire v490 servers running Apache [5] software to load balance web requests with a simple round robin scheme. The servers consist of 4 CPUs each, with 16 GB of RAM. The middle-tier consists of five clustered application servers running JBoss [8] software connected to a switch between the Apache servers. Each of the five servers is a SunFire v40z computer with 4 CPUs and 16 GB of memory. Two instances of the JBoss application web server run on each clustered machine. Finally, there are two high powered, 16 CPU, 32 GB SunFire 4800 servers running Oracle 10g and connected to the middle-tier with a 1gigabit Ethernet switch. The system has currently over 200 users and services thousands of different requests per day.

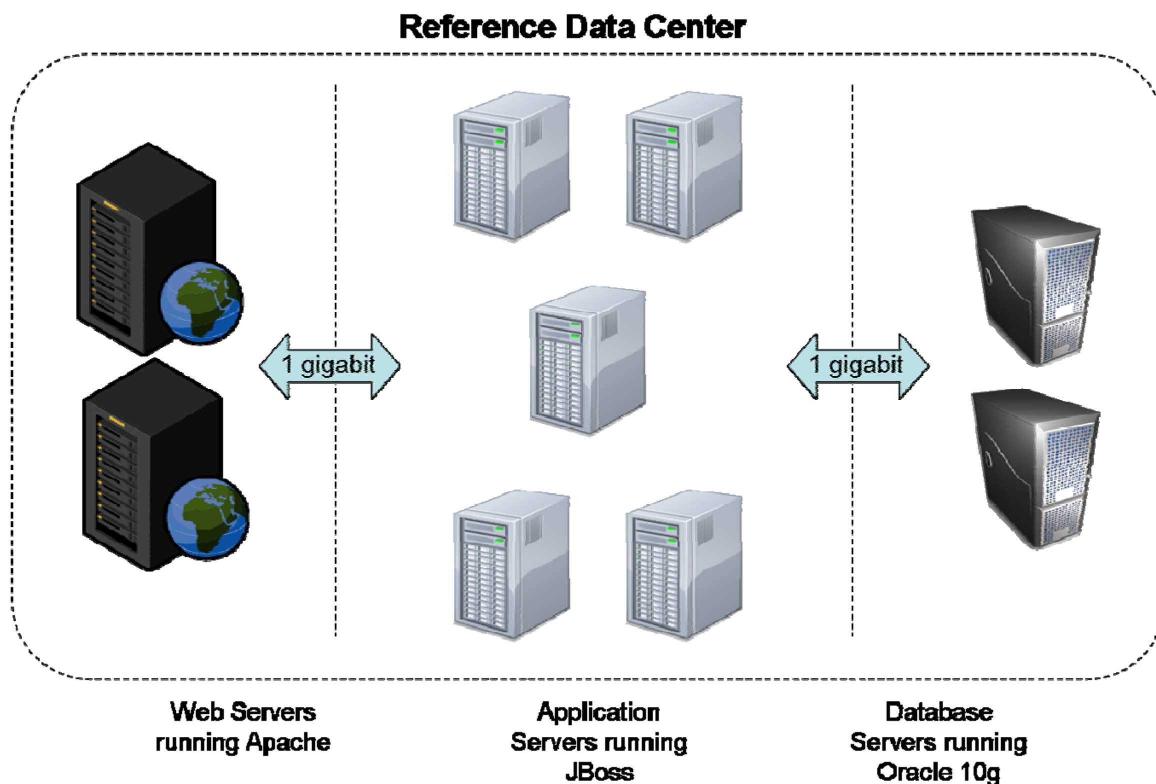


Figure 3-1: System architecture of reference data center

Several database instances are located in the persistence layer and are accessible through the data access service of the data center. Currently 14 databases are accessible with a plan to include more databases to the data center in the near future. Each database holds millions to a few billion, rows of data at any given time. A typical high volume data request from the data service will return hundreds of thousands of database records for analysis, potentially returning millions by the release of this thesis. Since each database schema holds a few hundred columns of data per record, it is evident that fast and efficient data transfer is required. The described architecture is the reference data center of our research.

3.2 High Volume Transfer of Relational Data

In this section we summarize a method of transferring large quantities of relational data utilizing a flexible binary Java object. Section 3.2.1 will provide an overview of present POJO methodologies and introduce a flexible technique for transferring relational data using Java. Section 3.2.2 illustrates a mapping technique of relational data to Java data types and finally, section 3.2.3 discusses how we enable dynamic length instantiation using XML descriptors.

3.2.1 Using Java Objects for Relational Data Transfer

POJOs are commonly used in services that access data from relational databases in data centers. ORM solutions [37, 45] map relational data from database tables into POJOs and getter and setter functions are used to access private members that are mapped to the fields in the database.

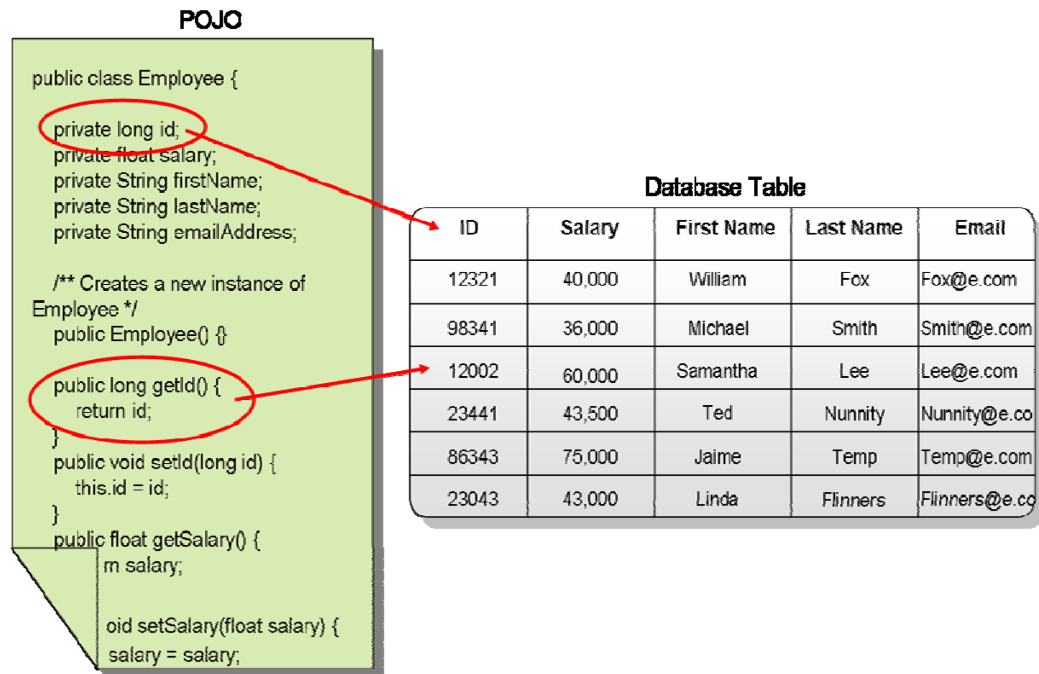


Figure 3-2: ORM example

POJOs provide a simple solution to the object/relational impedance mismatch when database schemas are relatively uncomplicated [56]. When massive quantities of data are desired with limited network resources, the POJO model does not perform well due to serialization overhead and run time specifications. POJOs are static and cannot be changed during run time; therefore if data is added, removed, or modified, the POJO approach will be limited. The goal is to improve both the serialization and extensibility shortfalls by creating a Java object that encompasses efficient binary serialization techniques and enables dynamic data mapping. To accomplish this, our approach first exploits relational data characteristics.

Since relational databases are the preferred models in data centers, we focus on relational data mapping to Java objects as our data transfer vehicle. Relational databases store data in forms of tables, in which a row constitutes a collection of data as a single object, and the columns of the table provide individual fields of that object. Relational database rows are exploited based on their characteristics; the arrangement of columns in the rows of a table will be identical and will maintain strict ordering. Therefore, each row in a particular database table is represented by the same Java object. Further more, any row in any database table will be a collection of columns that encompasses the same characteristics. Based on these observations, we create a Java object that can represent any row of data in a relational database table.

In this research, each record represents a particular entry of interest and the fields of the record represent characteristics of the entry. To encompass a small data footprint, we aim to directly map a database row to a Java object. The Java object is a simple class called JDatabaseRecord.java. JDatabaseRecord has a set of public arrays to hold data and represent the columns of a database row. The arrays representing fields of a record are accessed analogous to direct memory mapping. One row in the database is represented by one JDatabaseRecord. The array lengths stored in the transfer object are not static, but rather change dynamically based on the number of columns in a table. The columns are a subset of the entire set of columns a database row entails. Because our JDatabaseRecord class can represent any row of relational data, we introduce a dynamic allocation of array lengths to adhere to variable length requirements. By allowing server side processing to allocate the size of each array based on table specifications, we provide a flexible transfer object that stores only the exact data represented by a database record.

3.2.2 Mapping Java Primitive Values to Relational Data

It is essential that only the exact data represented in each returned field of a query is stored in the JDatabaseRecord object. Database columns can store numeric, character, or binary data in various formats, so the flexible Java object must be able to store all of these formats. We associate the data stored in a database column to Java primitive data types. Java has 7 primitive data types as well as the *String* data type to represent fundamental data formats.

Table 3-1: Java primitive data types

Java Data Type	Byte Representation	SQL Type Mapping
boolean	2 bytes	BOOL
char	2 bytes	CHAR
short	2 bytes	TINY
int	4 bytes	INTEGER
long	8 bytes	LONG, DATE, TIMESTAMP
float	8 bytes	FLOAT, DECIMAL
double	16 bytes	DOUBLE
String	Varying bytes	VARCHAR, NVARCHAR

Table 3-1 displays various primitive types in Java and the number of bytes used to represent each type. We refer to the Java type *String* as a Java primitive data type because it correlates with data stored in relational databases and is one of the 8 main types stored in JDatabaseRecord. By using these mappings to correspond to database column data, only the exact number of bytes needed to reconstruct relational data is stored in a JDatabaseRecord. In JDatabaseRecord, there are eight public arrays, each defined by one of these unique data types. Based on the columns requested during the retrieval of data,

each column is linked to a corresponding index in one of the eight arrays. All columns are thus grouped into one of eight arrays using the ColumnTypes.java enumeration. Each array length is dynamically instantiated based on the number of columns in a given group. The total of all eight array lengths make up the total number of fields that will be sent from the server to the client.

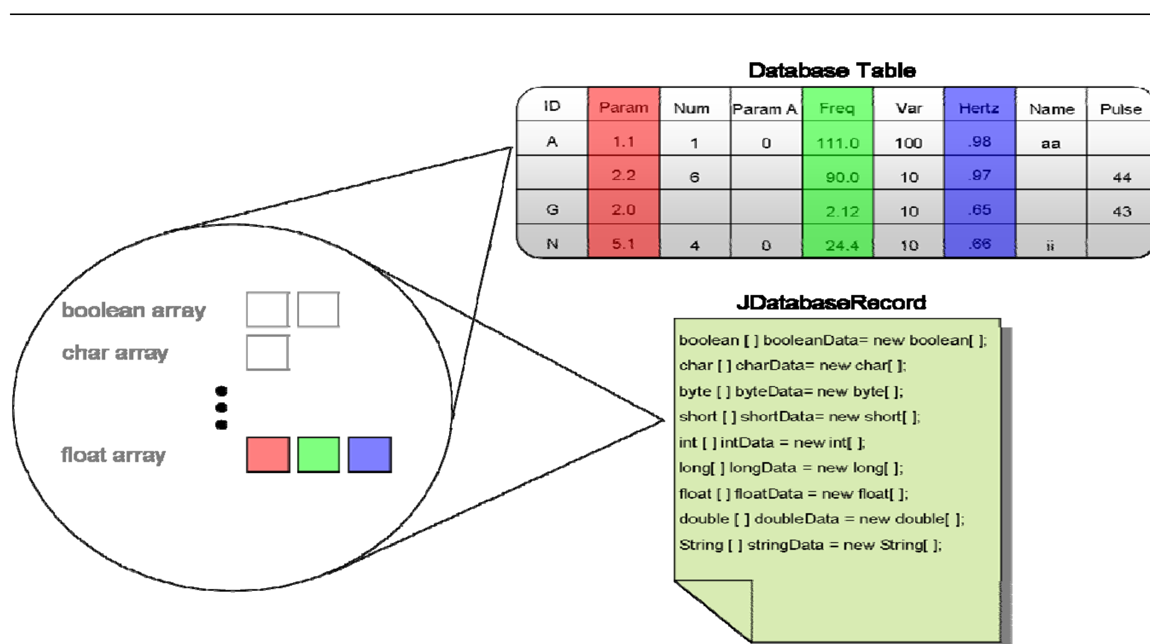


Figure 3-3: JDatabaseRecord primitive mapping

Figure 3-3 exemplifies how columns of a database table are mapped to arrays in JDatabaseRecord. The arrays in JDatabaseRecord act as direct memory mapping and are accessible using the publicly defined arrays on the client. The server processes all columns into specific groups, and maps the columns into the arrays of JDatabaseRecord. What results is a generic Java object that efficiently represents the fields of a particular row of a database table.

3.2.3 Using Metadata and XML for Dynamic Mapping

Databases represent data differently than Java. For example, a column in a table can be of SQL type LONGVARCHAR, but there is no obvious Java data type that maps directly to this SQL data type. Moreover, each database management system defines unique data types specific to their system (the reference data center utilizes Oracle DBMS so Oracle's SQL data types are noted throughout). Fortunately Java contains built in libraries that help bridge the gap between database SQL data type representations and Java primitive data types. JDBC is a library that provides a very useful API between Java and various database management systems [9]. An association is made between the JDatabaseRecord class and the columns of a schema through JDBC API functions. Column metadata is accessed by these API functions and are translated into an XML file specifying each available database table. The following figure depicts the described XML format.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<table>
<name>Relational Table 1<name/>
<columns>
  <column>
    <visual_name>Param A</visual_name>
    <column_name>a</column_name>
    <sql_type>NUMERIC</sql_type>
    <java_type>FLOAT</java_type>
    <precision>3</precision>
    <scale>2</scale>
  </column>
  <column>
    <visual_name>Type</visual_name>
    <column_name>type one</column_name>
    <sql_type>VARCHAR</sql_type>
    <java_type>STRING</java_type>
    <precision></precision>
    <scale></scale>
  </column>
</columns>
</table>
```

Figure 3-4: Database metadata XML example

Each column of a particular database schema is defined in the XML. The tags of the XML are specified as follows:

<name> - The name of an available database table
<**column**> - Defines a column of the specific database table
<visual_name> - Visual name of the column that can be used by client applications
<column_name> - The actual SQL column name representing this field in the table
<sql_type> - The database specific SQL data type that is used to represent this column
<java_type> - The Java primitive data type that is mapped using JDBC
<**precision**> - Database precision for numeric values
<**scale**> - Database scale used for numeric values

To generate the XML descriptor, the following steps are taken for each schema:

1. An XML descriptor file is created defining the name of the schema
2. The schema is queried from the system tables of the database and a ResultSetMetaData object is returned

3. For every column returned in the ResultSetMetaData object, metadata is written to the XML descriptor including visual name, column name, and SQL type
4. To map SQL data types to our JDatabaseRecord data types, a switch statement and the ColumnType.java enumeration is used
5. The ensuing ColumnType enumeration is written to the <java_type> tag

Using this procedure, when schemas are added, updated, or removed, the appropriate XML is regenerated. The binary Java object can adhere to these changes unlike a POJO. Therefore, when changes occur, the server side code (POJO objects that map directly to the database tables) does not have to be recompiled and redeployed with a recycle of the data center. Downtime can thus be avoided utilizing our methodology because dynamic changes will be realized by the XML regeneration after database changes are made.

Before any JDatabaseRecord is populated all request columns are grouped into separate lists specified by a ColumnType enumeration. Using the resultSet of the query, the corresponding column is read and using a switch statement the column is added to a list denoted by the number order of the column. See Appendix C for source code and details about this mapping. In this manner, each column is denoted by an index that correlates to the index returned from a SQL query. In effect, the *select* statement of a query orders the columns of a table, and this particular order is maintained in the lists. A hashmap that stores ColumnType enumerations is created, and lists of column indices are mapped for each key. Once complete, this information is used to generate each JDatabaseRecord. The following function declaration is used to store data from a database row into a JDatabaseRecord:

```
setValues(ResultSet rs, Hashmap<ColumnType, ArrrayList<String>> colMap, int cols);
```

The ResultSet is a JDBC object that contains several API functions to extract field information from a query. Several functions are available to supply access to data stored in the database as Java primitive data types. For example, ResultSet .getInt(2) will return a Java int value for the second column of the SQL query performed. Similarly, ResultSet .getFloat(10) will return a float value for the tenth column of the query. The second parameter of the setValues function is a hashmap defining each column mapping of a query. The hashmap allows access to the lists of each ColumnType to enable each transfer object to be populated from a corresponding ResultSet getter function. For all eight ColumnType enumerations, a ResultSet API function is called to provide Java data type representations; rs.getFloat(int columnIndex) maps to ColumnType.FLOAT data, rs.getBoolean(int columnIndex) maps to ColumnType.BOOLEAN data, etc. See Appendix A and Appendix B for the full JDatabaseRecord.java and ColumnType.java source code files respectively. After all columns are mapped, the JDatabaseRecord now stores the exact data from a row in the database and is ready to be serialized across the stream.

3.3 Serializing Binary Data Objects

This section discusses the serialization technique of transferring data represented in JDatabaseRecord.java across a stream. Section 3.3.1 summarizes Java Serialization pitfalls and introduces the Java Externalizable interface. Section 3.3.2 displays the data format structure utilized in our transfer object. Section 3.3.3 discusses the implementation optimizations to further decrease the amount of data sent across the network during

serialization. Finally, section 3.3.4 summarizes the entire client and server serialization process.

3.3.1 Java Externalizable Interface

Default Java serialization enables any Java object to be serialized as bytes and sent across the network. Byte data can vastly improve transmit performance compared to character representations of data such as XML. However, Java's serialization algorithm still places additional information needed to reconstruct trees of Java objects on a stream during transfer [52]. All members of a Java class send data representations as bytes; additionally objects send class description information which diminishes performance. Using the JDatabaseRecord object defined earlier, we can improve this algorithm by reducing the size of bytes transmitted across the network.

Java offers an additional interface that allows developers to specifically define how data should be sent when their Java objects are serialized. This interface is the Externalizable interface and contains two methods, *writeExternal*(ObjectOutput oo) and *readExternal*(ObjectInput oi). Using our JDatabaseRecord class we have implemented these two functions, and thus, use the Externalizable interface when serializing data. Drastic improvements are made to the serialization process because only explicit byte representations of stored data are transferred. Recall, we map database fields into fully accessible Java data type arrays. Because we are working with primitive data types, we do not need to keep further information needed to reconstruct a tree graph of nested Java objects. Therefore, we write only the data byte representation of our Java types to the

stream. We are able to reconstruct the data using the *readExternal* function implemented in the class, and the data is reconstructed into one of the eight data type arrays. This is accomplished without streaming additional class package information, thus, transferring significantly less data than the traditional serialization method. Additionally the employed algorithm enables dynamic lengths of data to be written and read from the stream, therefore modifications to the *writeExternal* and *readExternal* functions are not necessary when changes are introduced to the streamed data.

3.3.2 Binary Data Format

When byte data is written and read from a stream, the format of the data must be consistent for accurate object reconstruction. Simply, the order in which the bytes are read must follow the order in which the bytes are written. For example, if four bytes are written for an *int* followed by eight bytes written for a *long*, the *int* value must be read first, then the next eight bytes for the *long* value must be read. An exception will not be caught if the *long* value is read first because the proper number of total bytes is correctly read from the stream. There is no definitive way to determine if the data was reconstructed properly, especially if the total number of bytes is read but the bytes are read out of order. Therefore, it is important that ordering is strictly upheld when serializing data as bytes in the *writeExternal* and *readExternal* functions. To adhere to strict ordering, a specific format is utilized in the JDatabaseRecord.java class to retain consistency and allow data to be reconstructed during serialization. Since eight data type arrays are stored in JDatabaseRecord, the following order is upheld:

writeExternal()
1.) boolean 2.) char 3.) short 4.) int 5.) long 6.) float 7.) double 8.) String

readExternal()
1.) boolean 2.) char 3.) short 4.) int 5.) long 6.) float 7.) double 8.) String

Every JDatabaseRecord object writes and reads data in this order. A *for* loop writes and reads the data depending on the length of each array. Because dynamic length array construction is exploited, we cannot statically define a length used in each loop to determine how many bytes are read. Consequently, the following solution is applied.

To overcome issues surrounding dynamic length construction of serialized objects, JDatabaseRecord utilizes common information known to both the client and server to help during serialization. When a request is made from a client connected to the data center, the client application either statically knows the number of fields retrieved, or dynamically requests specific fields to be returned. From this information, each data type length is inferred. Each field is grouped into a specific ColumnType enumeration, after all fields are grouped, the length of each array can be stored. To store dynamic length information, Java's system property API is utilized. When Java objects are rebuilt on the client during serialization, the default, no parameter constructor is called. During client reconstruction, Java uses reflection to instantiate each JDatabaseRecord. The default, no parameter constructor is used to obtain variable length information for each Java primitive type. The constructor reads this information from Java system properties. As a result, when the *readExternal* function is called, each data type length is applied when reading bytes from the stream. *For* loops are formulated to read the exact number of

bytes and a resulting JDatabaseRecord is reconstructed. All transfer objects exploit saved length information to enable variable length transfer of data.

In contrast, another methodology for sending dynamic length data is to send length information on the stream regarding the number of each data type [13]. This could be done in the same order in which data is streamed in the proposed technique; before each data element of an array is written to the stream, the length of the array could be placed on the stream. With this approach, an extra 16 bytes of data are written for each JDatabaseRecord (8 public arrays and lengths are represented by 2 bytes each).

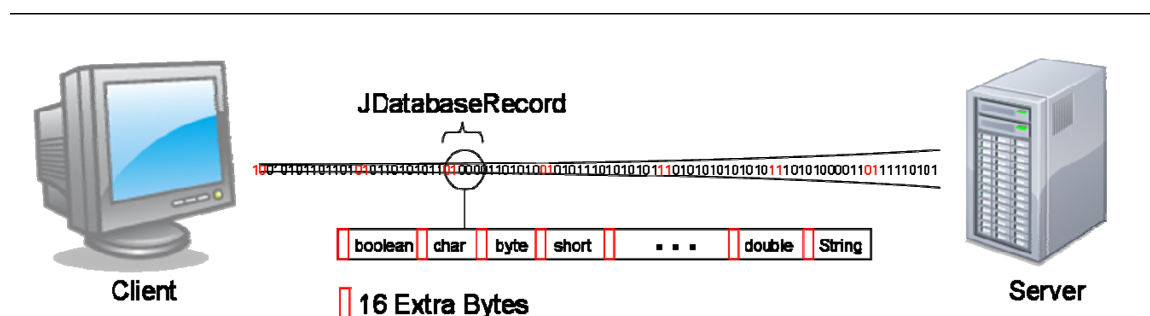


Figure 3-5: Illustration of extra bytes from contrasting data format

The number of additional bytes may seem minimal and may not seem to affect performance; however, the goal is to develop a fast and flexible transfer object of relational data for high volumes of transfers. If hundred of thousands to millions of transfer objects are sent, an additional 16 million bytes (16mb) of redundant data will be added to the stream. Taking this into consideration, we levy the requirement that the client must infer the length of each data type array before a request is sent to the data center. Thus, rebuilding data on the client will be used to improve performance.

3.3.3 Additional Techniques for Faster Transfer

We employ a few techniques to further reduce the number of bytes that are written to the stream during serialization. First, a reduction technique prevents redundant data to be serialized over the wire. Figure 3-6 defines an example database table with 10 columns of data, each storing data as doubles values.

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10
0.0	99.9	1.1	11.1	0.1	0.0	211.1	.98	13.13	43.1
0.0	1.23	0.0	02.3	0.0	90.0	20.0	0.0	0.0	0.0
234.0	0.0	2.0	22.0	0.0	0.0	0.0	.65	0.0	23.0
0.0	111.0	0.0	0.0	0.909	24.4	0.0	.66	0.0	10.0

Figure 3-6: Database fields double value representations

In this example, many of the columns store the same numeric value zero. While some of the fields hold unique values, a majority of the fields are similar. We can exploit this homogenous characteristic and reduce the amount of data to transfer. In this example, each reduction of data represents a *double* value and saves 16 bytes. A substantial amount of bytes are saved using this reduction process.

This method is employed for two values stored in the reference databases. By default, JDatabaseRecord.java will not write null or zero data stored by a requested field during transfer. The decision to choose the value zero is two fold. First, an assumption is made that the value zero is the most common redundant numerical value stored in a database. The value zero is used for integers, floats, doubles, shorts, bytes, monetary

values, and other numeric fields, and there is a greater probability that zero will be stored in these fields. Second, in the currently deployed data center, many fields contain the value zero and this characteristic should be exploited. The decision to use the value null is based on the ability of current DBMS columns to be declared nullable. When a database field is not required to have a value, the value null is used instead. The potential serialization improvement becomes immense when many fields retain one of these characteristics.

To represent null or zero data, two additional arrays are created in JDatabaseRecord.java and are written and read on the stream before the other data types are processed. The arrays are simple Boolean arrays that are used to represent whether or not a field is null or zero. The length of the array is the size of the total number of columns desired during a request for data. By default, all values in the two arrays are false. If a column is null or zero, the index in the respective array that represents the column is set to true. Fortunately, when the JDatabaseRecord.java class is instantiated in the business logic layer of the data center, it is easy to determine if a field is null or zero using the ResultSet API. Recall that every column is mapped to one of eight data types (represented by the ColumnType enumeration) in the transfer object and a corresponding ResultSet API call is made to obtain the java data for this particular byte representation. Inside the *setValues* function of JDatabaseRecord, the ResultSet function *wasNull*() is called before data is stored in a corresponding array. Either true or false is returned from the function representing a null or non-null field. If the result is true, a global index is used and the index of the null array is set to true. The same applies when checking zero data. If zero data is found using a simple evaluation check, the analogous value in the

zero array is set to true using a different global index. The following pseudocode describes the *setValues* function.

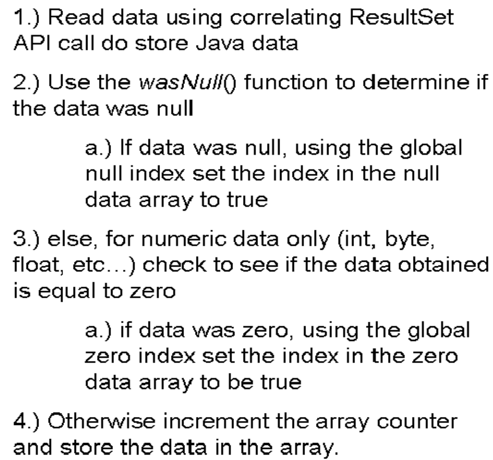


Figure 3-7: JDatabaseRecord setValues() pseudo-code

When null or zero data is set for a given index, resultant data should not be stored in an array and should not be serialized. During instantiation of each array, the length of the array is obtained by the columnMapping parameter passed into *setValues*. Eight counters are initialized to zero and represent the number of values currently stored in each array. If a non-null or non-zero field is obtained, the data is stored in the array using the counter as the current index and the counter is incremented. Contrary, if null data or zero data is obtained, the counter is not incremented. After all values of a specific type are obtained, the length of the array will be defined by the counter value. Therefore, inside the writeExternal function, the individual counter for each array type represents the actual

length of the array. Only non-null and non-zero data is thus stored in the array and the counter represents this data's length. Reverting to the previous example, if 8 out of the 10 fields of the table were zero or null, then only 2 fields would have to be serialized. However, the null and zero arrays need to be sent across the stream defining which of the data is null or zero. Our second approach for reducing the data is defined for this case.

Each Java Boolean value is represented with 2 bytes of data when serialized. A Boolean value simply represents true or false, therefore, Boolean data is characterized as single bits to reduce the amount of data serialized. Inside the *writeExternal* and *readExternal* functions, Boolean data is processed by storing eight Boolean values in a single byte. Therefore, the null and zero arrays are serialized as single bits per field, which drastically reduces the overall number of bytes that would have been transmitted using default Java serialization. The following figure depicts the algorithm applied for all Boolean data in JDatabaseRecord.

1.) obtain the length of the null data array

2.) divide the total length by 8 (8 bits per byte) and store the ceiling of this number (i.e. 10 / 8 = 2) as the total number of bytes

3.) loop through the total number of bytes stored from above

    a.) Loop through 8 times (8 bits per byte)

        i.) For each item in the loop, set the specific bit to be zero (false) or one (true) by checking the null data array using the global index.

        ii.) increment the global index

    b.) write the byte to the stream

4.) Repeat steps 1-3 with the zero data array

Figure 3-8: Single Bit Algorithm (SBA) for transferring zero and null data

We refer to this approach as the Single Bit Algorithm (SBA). By adding the null and zero arrays to help reduce redundant data, we only add ((2 x Num Fields) / 8) bytes of data to the stream. Additionally, any Boolean values obtained from the ResultSet can also make use of this algorithm. Referring to the previous example, if 8 out of the 10 fields are null or zero, and each null and zero array can be represented as 1 byte (8 fields each represented by a bit), a total of 128 – 2 = 126 bytes are saved per record when this data is serialized. Refer to the writeExternal function in Appendix A for details regarding the implementation.

3.3.4 The JDatabaseRecord Serialization Process

JDatabaseRecord is a Java object that enables fast binary serialization of data using the Externalizable interface. Inside the *setValues* function of JDatabaseRecord.java, null data, zero data, and specific data representations are stored in public accessible arrays. Global counters represent the actual number of values stored per array configured from the data obtained from the ResultSet and the offset computed from the null and zero arrays. After storing all data type representations using the *setValues* function, the *writeExternal* function writes the stored data as bytes to a stream. Function *writeExternal* utilizes SBA to reduce the number of transferred bytes for Boolean data. SBA is applied to the null, zero, and Boolean public arrays of JDatabaseRecord.java. Next, using global counters, the other 7 arrays of data are written to the stream in proceeding order using simple the ObjectOutput API and *for* loops. Data is then transmitted across the stream as bytes to a client machine.

Once on the client, the *readExternal* function reads all bytes from the stream similarly. First, the serialization process invokes the default JDatabaseRecord constructor where information is obtained providing exact lengths of the null and zero arrays (the total number of fields requested), as well as potential lengths of each data type array. Inside *readExternal*, SBA is applied to the null, zero, and Boolean data arrays. Next, each data type is read in the exact order that it was written using a simple *for* loop and a newly computed array length. Based on the null and zero data, the length of each data type array is calculated before bytes are read. This is accomplished with a simple algorithm illustrated in Figure 3-9.

1.) store the default length of the array to a variable

2.) initialize a size variable to be zero

3.) loop while length variable is greater than zero

    a.) check null and zero arrays (using global indices).

        i.) If null and zero arrays are not set, increment size variable

    b.) decrement length of the array

    c.) increment global null and zero index

4.) Return size variable which represents the true size of the array.

Figure 3-9: Pseudo-code for storing data in a JDatabaseRecord

Each array length is recalculated by stepping through the null and zero arrays to determine what column index is either null or zero. This enables accurate reconstruction of the JDatabaseRecord through the client serialization procedure.

Our methodology utilizes the Externalizable interface as well as XML descriptors to develop a flexible and efficient binary transfer object. Relying on relational data characteristics, a simple yet dynamic Java object is implemented to map data from relational database table columns into Java primitive data types. Additionally, specific implementation techniques are employed to drastically reduce the amount of bytes streamed during the serialization process. As a result, significant reductions in latency are achieved when transmitting data over traditional HTTP protocols.

3.4 JFlexBOT: Java Flexible Binary Object Transfer

Java Flexible Binary Object Transfer (JFlexBOT) is the described methodology of sending Java data type byte representations, mapped from SQL data types stored in relational database fields, over the network for efficient and flexible binary transfer. Section 3.4.1 overviews the relational data access service currently deployed in the three-tiered data center described in section 3.1.

3.4.1 Developing a Relational Data Access Service with JFlexBOT

JFlexBOT engrosses an efficient approach to serialize column fields of a database across a stream. Currently, JFlexBOT is implemented in an operational data center that enables access to large amounts of analytical data. The service entails a request and response Java object in which a database query is sent to the data center and a list of JDatabaseRecords are returned. Once returned, the records are used to represent data on several first echelon visualization tools represented by a Java desktop web application. The following is a high level overview of this service.

The data access service allows queries to be formulated to different databases through a simple XML request specifying the database to connect to, the columns from the database to be returned, and the filters which construct a SQL statement. The request uses information gathered about the available databases and the fields of the database tables. This is accomplished in the current data center with a simple web service, which returns XML for each accessible database (refer to section 3.2.3 which explains how the XML is generated on the server). Thus, during client startup a client obtains XML

information and stores the descriptors in memory for the runtime of the application. Using the **<column_name>** tag obtained with the XML, requests are generated defining the desired fields to return. The fields are then grouped into ColumnType lists to compute the potential max length of each array. The XML file, ColumnType enumeration, and source code provided in Appendix C are all used for grouping. Lastly, before the request is sent, system properties are stored defining the potential max length of each array in JDatabaseRecord. This enables the client to properly obtain these results and reconstruct each JDatabaseRecord during serialization (see the JDatabaseRecord.java constructor in Appendix A for implementation specifics).

During data center start up, each XML descriptor is created for all accessible databases. For a given request to the server, a column mapping is computed using the XML descriptors and column type enumerations. The mapping needs to only be created once per request because all JDatabaseRecords that are returned maintain the same format and use the same column mapping. A SQL query is then initiated and a resultant JDBC ResultSet is obtained. For each record returned in the ResultSet, a transfer object is instantiated using the single parameter constructor of JDatabaseRecord.java. By calling the single parameter constructor, a new Java object is created without reading system properties to obtain array length information. Rather, the ResultSet and previously constructed column mapping are passed into the *setValues* function where the data is obtained from the ResultSet and stored in each array. After processing is complete in the *setValues* function, the JDatabaseRecord is added to a Java list that is returned from the server. Once all records are created, the list is returned to the client invoking the Java

serialization process and calling the *writeExternal* method to write the bytes to the stream.

The client serialization process is invoked once the records are written to the stream and passed over the network. Initially the default no parameter constructor is invoked implicitly; which stores the potential max lengths of the arrays of the transfer object. Next, the *readExternal* function is called and the record is reconstructed with the data. Finally, after all objects are processed, the client application can access the list of records and begin using the data. Figure 3-10 depicts the data access service process.

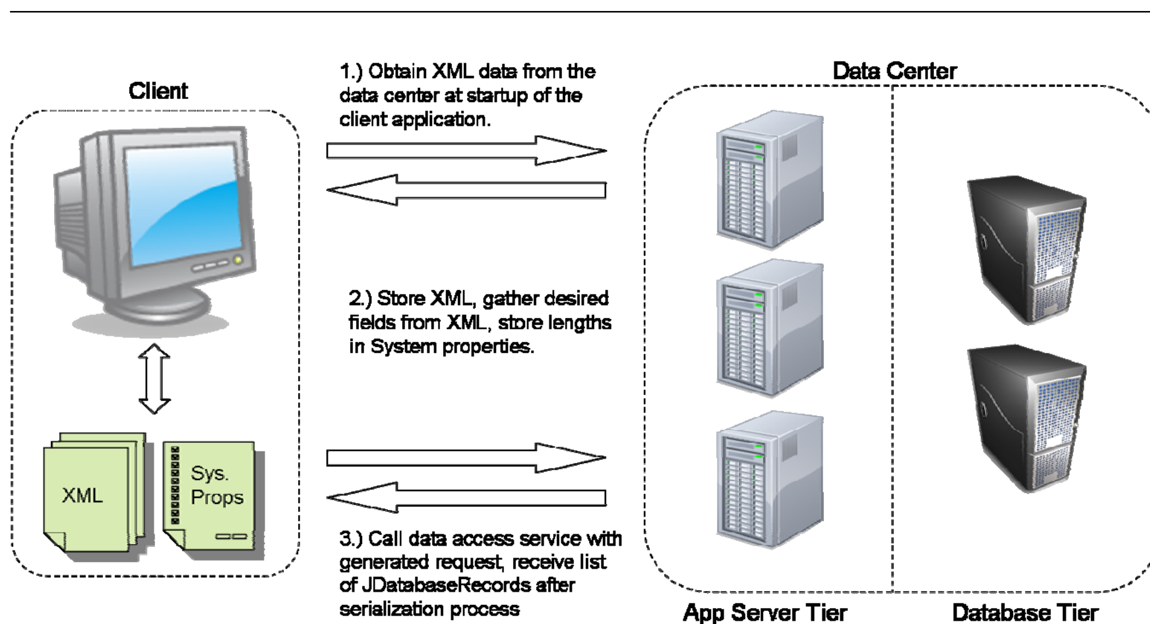


Figure 3-10: Data center data access service workflow with JFlexBOT approach

This workflow illustrates service oriented architecture methodologies that help build a system resolving the conventional POJO maintenance issues. With the JFlexBOT approach, no longer do backend server side changes affect data center downtime.

The XML descriptors that are accessed from both the client and server allow dynamic changes to the database without penalizing the data center with maintenance overhead. Also, client applications are able to define precisely what data they would like to receive from the service. Hence, a more robust service oriented architecture is implemented. The evaluation of this service is presented in the next chapter.

Chapter 4

Experiments and Results

The following chapter documents experiments and results between the proposed Java binary flexible transfer methodology and a generic POJO transfer approach. The experiments are conducted on a deployed data center accessible by over two hundred end users. By using an operational system, accurate metrics are obtained for comparison. All experiments gather metrics to determine the primary latency and throughput of the two methodologies and comparisons are concluded based on the findings.

4.1 System Setup

The currently deployed system described in Chapter 3, Section 1 is used to conduct the experiments for the previously outlined approaches. To run the experiments, a remote client machine was connected to a classified network accessible to the data center. The client machine used was a SunFire v240 workstation running the Solaris 5.9 operating system. The WAN network connected from the client workstation to the data center transmitted at roughly 333Kbps during the time of the experiments. The experiments were run during normal data center workload so that transmission latency and throughput could be recorded accurately. Before running the tests, the data center was adapted to maintain both the JFlexBOT approach and the POJO approach when processing requests. A test web archive file (WAR) was deployed to the operational data

center, allowing either approach to be used when gather data. Additionally, testing code was added to the data center to enable metrics to be gathered for the experiments.

To perform the experiments, a test application written in the Java programming language was developed. This application administered the experiments and collected client side metrics. The application sent HTTPS requests to the data center and received results through an HTTPS response. The number of data records returned was limited to 50,000 per response so that HTTPS timeout errors would not occur during long delays. The HTTPS protocol was utilized for the following three reasons:

1. Based on the sensitive nature of the data that was transmitted across the network, secure connections are required.
2. The current data center implementation utilizes the HTTPS protocol.
3. One of the goals of the research was to utilize traditional web protocols when transferring data. Because HTTPS adds additional communication overhead during transmit, it was decided to use HTTPS over HTTP.

The client application performed several experiments with both approaches, and both the client and server metrics are presented in this chapter.

4.2 Experimental Procedure and Metrics Definition

Java flexible binary object transfer provides a generic approach for transmitting variable length relational data and aims to reduce the number of bytes that need to be transferred. To provide a comparison between JFlexBOT and the POJO methodologies, metrics are obtained from several experiments run on the deployed data center. The experiments are run several times to enable averaged metrics to be gathered for

comparison. Field declarations are used to return variable length record sizes based on the two methods discussed. Three distinct record field declarations are reviewed and the field declaration sizes for each record are defined as:

1. Small Record – 15 columns
2. Medium Record – 50 columns
3. Large Record – 150 columns

The three record sizes provide a good replica of data retrieval in the working system. An analyst usually requests between 20 and 80 fields for data analysis, therefore, these three size declarations provide an adequate range for accurate results.

The experiment conducts four queries to retrieve high volumes of data from the data center. The four queries have total record sizes of 25000, 100000, 250000, and 1000000. Therefore, for each given field size declaration (small, medium, and large), four queries of 25000, 100000, 250000, and 1000000 records are completed. The experiments are conducted from small to large size declaration, varying the transfer methodology. For example, the first experiment requests 25000 records with 15 columns of data for each record using the POJO approach. Three separate experiments are completed for this size declaration and number of records for the POJO approach. After the experiments are complete, the same size declaration and number of record experiments are performed using the JFlexBOT methodology. Once the small size declaration experiment is performed for both the POJO and JFlexBOT approaches, the next series of experiments for 25000 records with 50 columns per record is completed. This ordering is followed for

all experiments resulting in 72 total experiments. Table 4-1 illustrates a definition matrix for the experiments.

Table 4-1: Experiment Matrix

	Plain Old Java Object (POJO)			**Java Flexible Binary Object Transfer (JFlexBOT)**		
	15 Columns	**50 Columns**	**150 Columns**	**15 Columns**	**50 Columns**	**150 Columns**
25,000 Records	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs
100,000 Records	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs
250,000 Records	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs
1,000,000 Records	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs	3 Runs

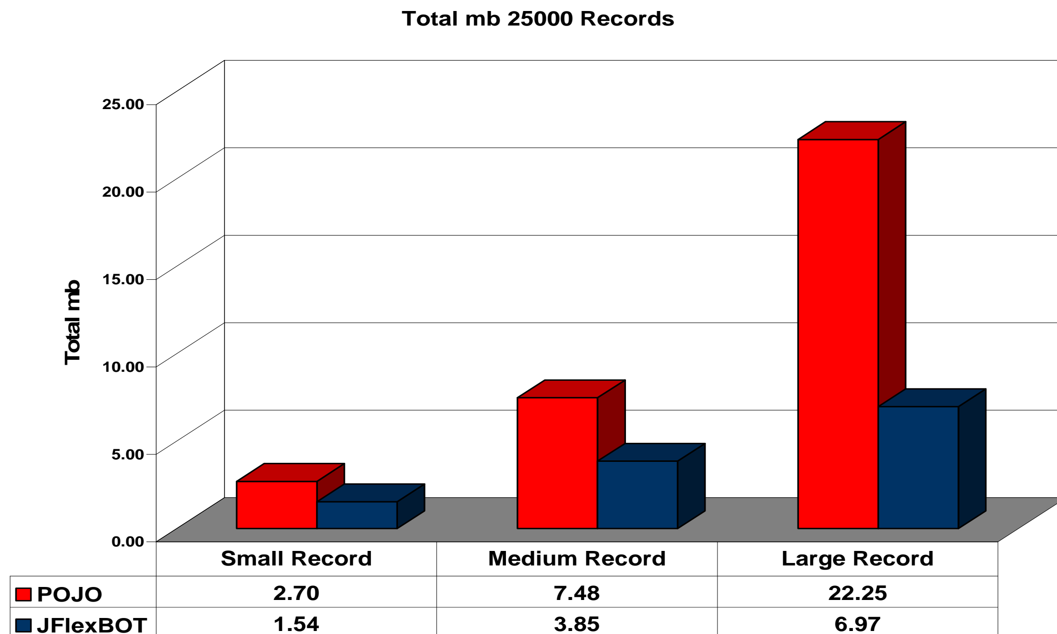
Three experiments are conducted for each field size declaration and for each size of records returned. Time constraints and system specification limited the number of experiments that could be conducted. Nevertheless, the replication of the experiments provided ample results for comparison.

To compare the two approaches, several metrics are gathered during the administration of the experiments. The first metric obtained includes the number of bytes transmitted across the stream. Both the total transmission sizes, as well as the number of bytes per record, are measured for all experiments. Timing metrics are also obtained during the experiments including read and write serialization times, server processing time, and total request time. The serialization times are calculated by summing the total time it takes for each record to be serialized on the server (write time) and the total time for each record to de-serialized or read from the stream (read time) on the client. The server processing time is defined as the time (*T)* taken to read data values from the

database and store these values for transfer. The POJO methodology utilizes the getter functions of the POJO to map each field in the database. The JFlexBOT methodology dynamically maps data by using the JDBC SQL types obtained from the ResultSet of the query. Therefore, the server processing time is measured by the time each technique takes to map data, and accumulating this time for each record. Additional metrics including transfer time and throughput, which is defined as the number of records transferred per second, are also presented.

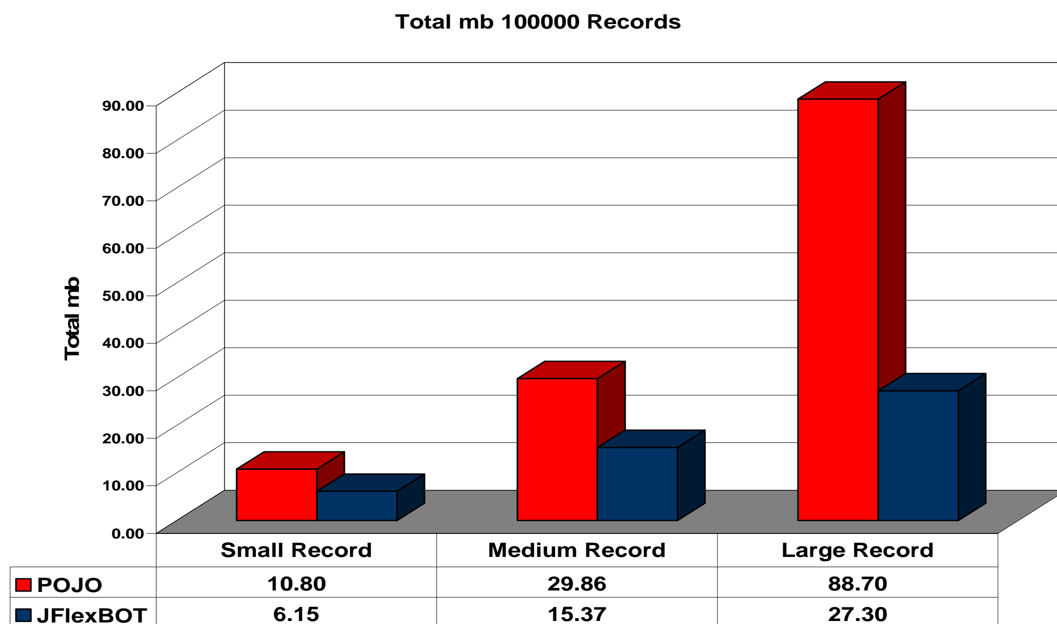
4.3 Results and Comparison

The metrics obtained from the performed experiments can be found in Appendices D and E. Appendix D presents tables for all metrics of the POJO experiments, while Appendix E provides tables documenting the metrics for the JFlexBOT experiments. Using the data provided in the Appendices, clear comparisons are performed between the two approaches. The first comparison examines the total size of data transferred during each experiment and the number of bytes per record. The following charts convey the total bytes transferred during all experiments.



	Small Record	Medium Record	Large Record
POJO	2.70	7.48	22.25
JFlexBOT	1.54	3.85	6.97

Figure 4-1: Comparison of 25000 records



	Small Record	Medium Record	Large Record
POJO	10.80	29.86	88.70
JFlexBOT	6.15	15.37	27.30

Figure 4-2: Comparison of 100000 records

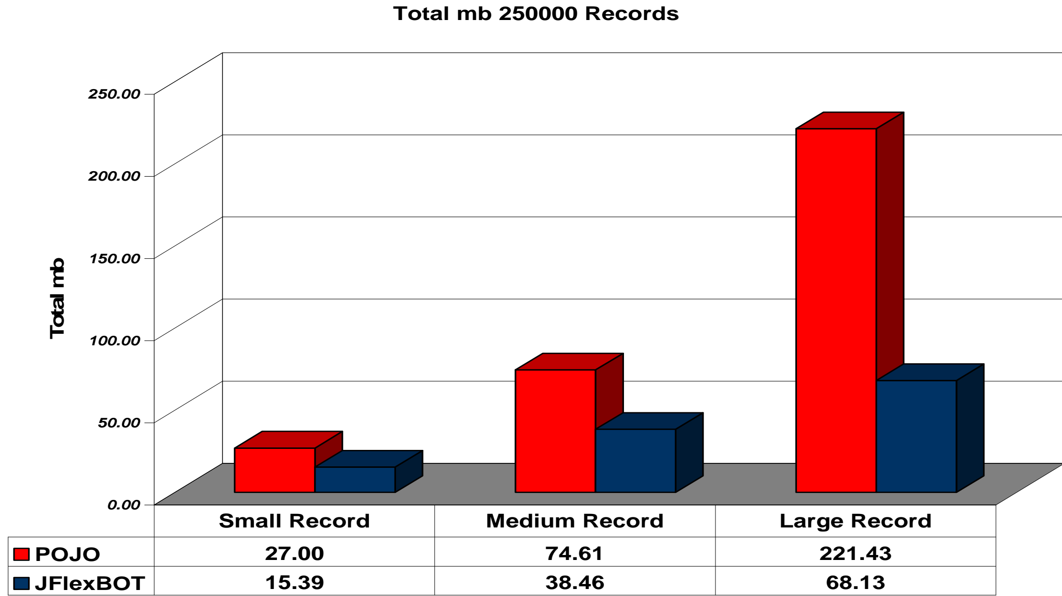


Figure 4-3: Comparison of 250000 records

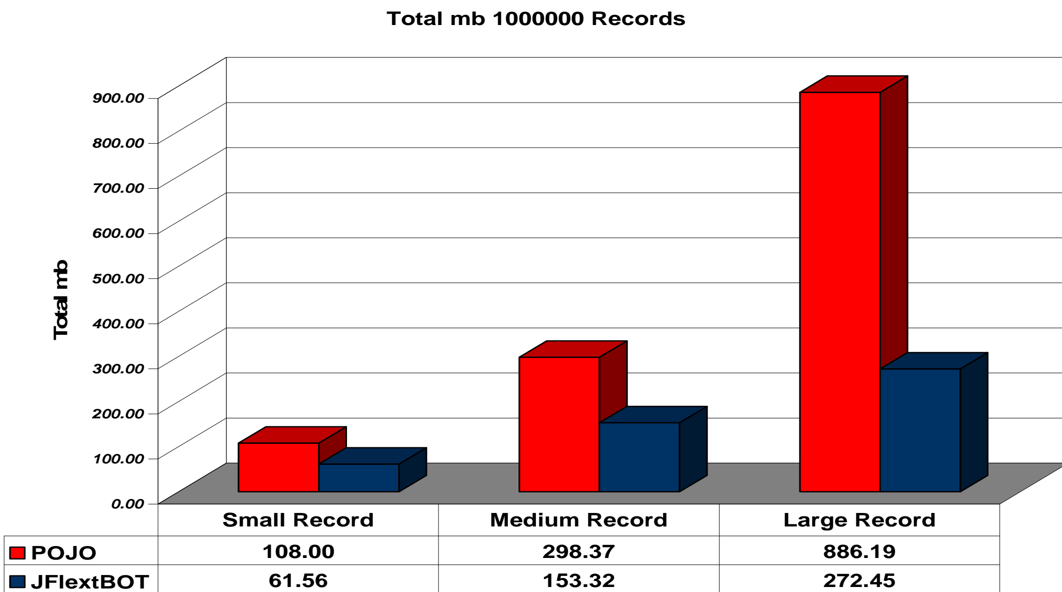


Figure 4-4: Comparison of 1000000 records

The results illustrate that a 43% size reduction is accomplished with the JFlexBOT approach with small record sizes. The reduction increases to 49% and 69% for medium and large record sizes respectively. This can be contributed to the serialization implementation differences between the two methods and the reduction in bytes transmitted on the stream from the Single Bit Algorithm of the JFlexBOT approach. The experiment requesting 1 million records with 150 columns per record reduces the overall data size from 886.19 to 272.45 using these techniques. For all experiments, reductions in size become more evident as the number of fields per record increases and the number of records increases. The characteristics of the relational data (zero and null values) along with the Single Bit Algorithm are mostly responsible for the size reduction. However, the externalizable implementation of JFlexBOT also reduces the number of bytes transmitted by only sending primitive representations of the data on the stream. A similar trend is obtained from the average bytes per record metric.

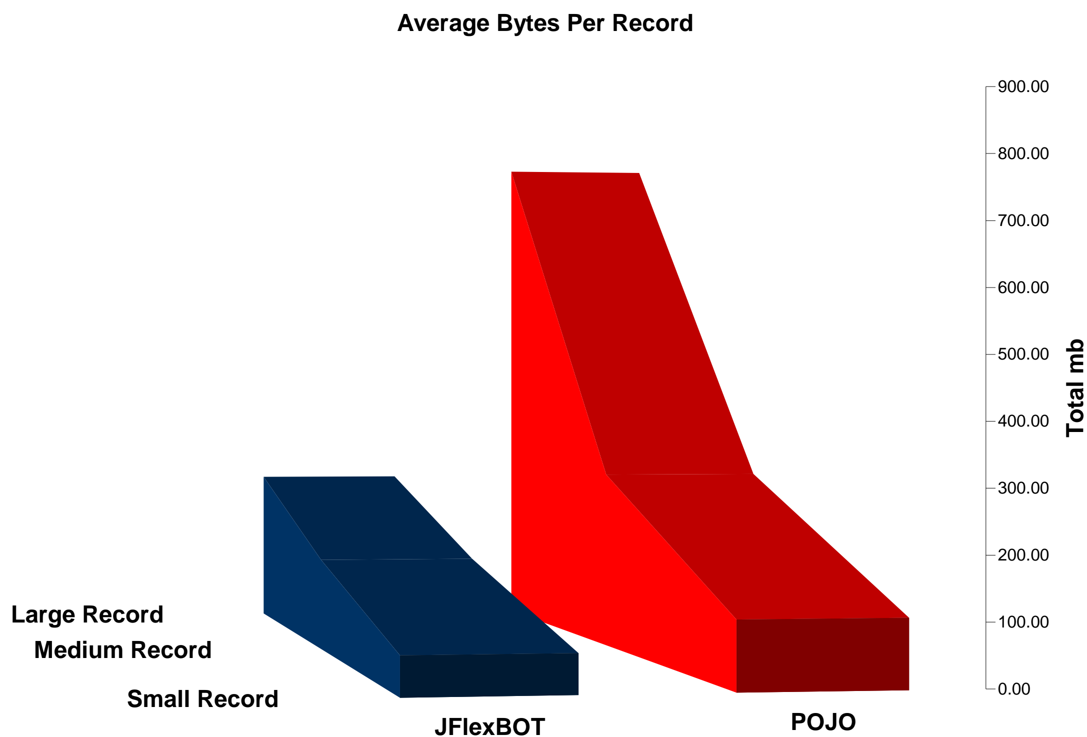


Figure 4-5: Average bytes per record

These metrics correspond to a 43%, 49%, and 69% reduction in size for the small, medium, and large records respectively.

Another measure of performance is obtained from network throughput. Figure 4-6 displays the throughput for the small and large records of four different experiments.

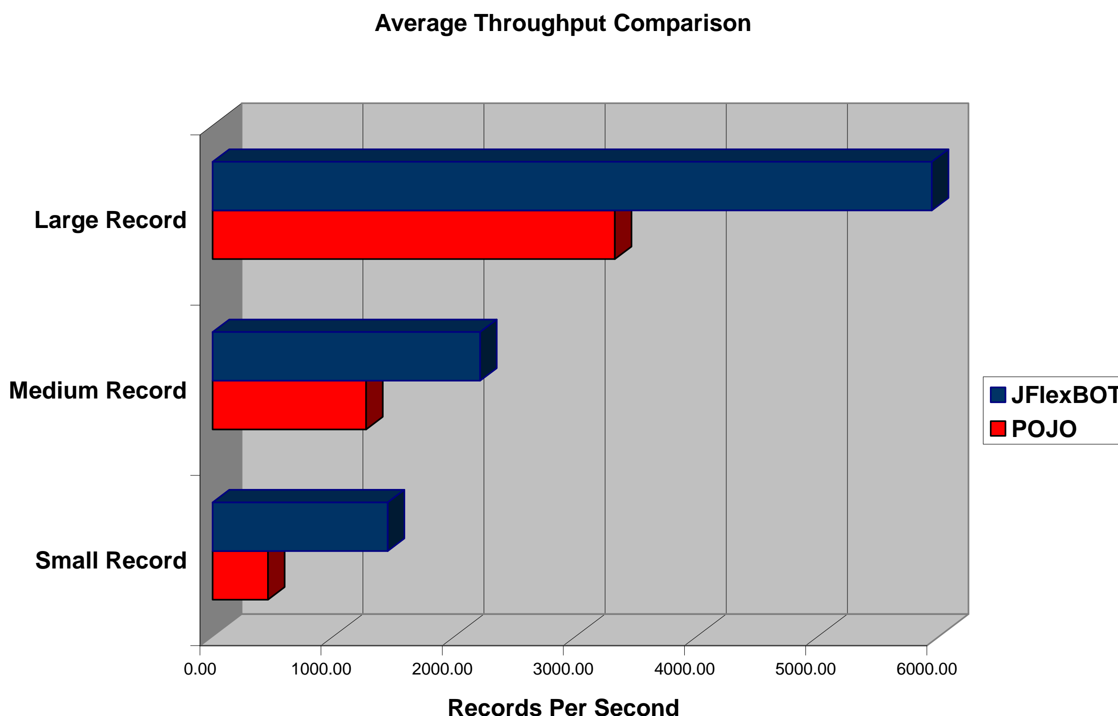
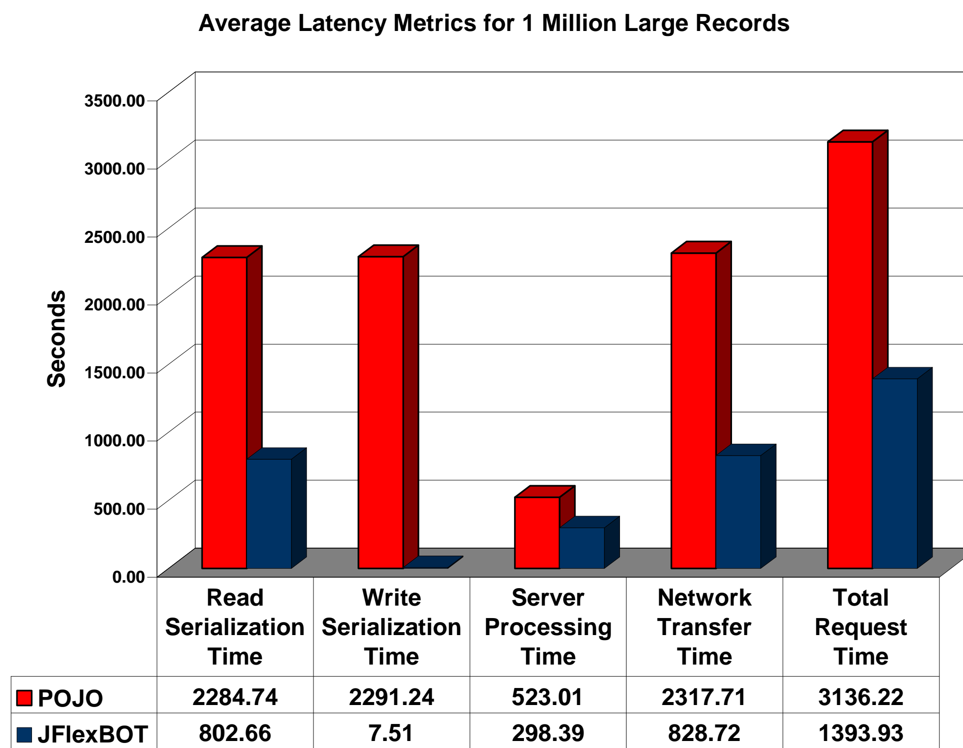


Figure 4-6: Average throughput for small, medium, and large records

The throughput for the experiments measures the maximum number of records per second that are returned to the client. The data provides an accurate performance measurement for the varying techniques. Figure 4-6 reveals that JFlexBOT maintains better throughput in both the small and large record experiments, which increases to almost 300% for large field size declaration experiments. The reduction in write and read serialization time plays an important role in increased throughput.

Further comparison can be deduced from the total transfer and serialization times of the experiments. Transfer times are bound by network performance, but the following chart helps disseminate the improvement of JFlexBOT over the POJO approach.



	Read Serialization Time	Write Serialization Time	Server Processing Time	Network Transfer Time	Total Request Time
POJO	2284.74	2291.24	523.01	2317.71	3136.22
JFlexBOT	802.66	7.51	298.39	828.72	1393.93

Figure 4-7: Experiment latency metrics for 1 million large records

Figure 4-7 shows improvement in all gathered metrics for the JFlexBOT methodology. The read and write serialization times are improved by 65% and 99% respectively. Because the read and write serialization process is not synchronous and records can be serialized during the transmission of other records, JFlexBOT enables data to be written to the stream quicker. This enables a 64% enhancement in network transfer time compared to the POJO technique. The augmented serialization processes result in diminished network transfer times and reduced total transfer time. Fig. 4-7 confirms that JFlexBOT can improve real world data request performance by over 55%.

The reduction in serialization time is partly due to the implementation of Java's Externalizable interface and the setValues implementation of JDatabaseRecord.java. When data from a ResultSet is mapped to the JDatabaseRecord in the setValues function, the data is written directly into a memory mapped byte buffer (refer to Appendix A for details). Consequently, the writeExternal function simply writes the byte buffer directly to the output stream, eliminating further data processing. Null and zero data are not written to the stream and, therefore, the size of the serialized records is decreased. By using these techniques, part of the write serialization time is realized in the setValues function. In contrast, the POJO technique maps all data obtained from a database using the object's setter functions. All data, zero and null values included, are written to the stream. Furthermore, the procedure writes variable name descriptors, java class package definitions, and byte representations of the data to the stream. The difference between the methods results in a 99% improvement for the JFlexBOT approach.

The time gathered to read data from the database and store the data in each transfer object is labeled as the server processing time. As previously stated, the JFlexBOT approach exploits a byte buffer to store the database data before transfer, reducing the time needed for processing in the writeExternal function of JDatabaseRecord. This implementation also reduces the server processing time because null and zero data are not stored in the byte buffer, and the dynamic mapping of our approach has correctly mapped Java primitive types to the resultant SQL fields of the database. The POJO methodology does not guarantee the smallest byte representation of data unless the developer of each POJO consciously limits the primitive type for each

field in the database. This can be a time consuming task and is prone to mistakes. This approach results in less than preferred performance.

The aforementioned analysis provides concrete evidence that the JFlexBOT approach improves performance and reduces latency of high volume data access over web protocols. The results conclude that JFlexBOT impressively reduces the write serialization time, dramatically reduces the size transferred for each record, and increases the throughput of the data center's data access service. As more data is requested from the data center and at faster rates, the improvements of JFlexBOT become even more evident. Additionally, even though measurements were not provided for the reduction in implementation overhead when exploiting JFlexBOT techniques, the dynamic data transfer methodology is capable of drastically reducing the software lifecycle.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

More data centers are becoming large data warehouses to encompass an array of data services to end users. With the growing demand of data analysis, it is becoming prevalent that high volume access is desired. The three-tiered layered approach of present data centers provides a good foundation for these systems. However, the demand for data access is emphasizing the poor performance that is occurring between the client machines and the first layer of these data centers. To alleviate the latency, solutions must be developed to improve the performance and allow more data to be transferred than traditional Java approaches.

A Java flexible binary object transfer (JFlexBOT) methodology was developed in this research to improve performance of high volume data transfers. Our approach enabled large quantities of relational data to be efficiently transferred from massive data centers to web applications using traditional web protocols such as HTTPS. Along with an efficient data transfer solution, the methodology provides a generic transport format of relational data which effectively allows any relational row of data to be mapped to our Java binary object. The proposed technique allows binary data to be streamed efficiently because of such implemented techniques as the ‘Single Bit Algorithm’ and dynamic data mapping. By representing null, zero, and Boolean data as single bits, drastic size

improvements were accomplished. The goal of the research was to decrease latency, increase throughput, and reduce transfer size of data, and all three of these goals were accomplished with JFlexBOT. Additionally, with the improvement in performance a reduction in data center energy output can be obtained. Less data is transferred and more efficient processing is accomplished on the server, both result in more efficient data center energy consumption.

To validate performance improvements, JFlexBOT was compared with a well known Java approach called plain old java objects (POJO). POJOs are data storage objects that can be serialized and transferred as binary data from a server to a client. Many data centers encompass and transmit data with HTTP requests using this POJO methodology. For our research, 72 experiments were conducted that gathered metrics on the two approaches. For each approach, three distinct record sizes, and four varying response size requests were conducted. The four requests included 25000, 100000, 250000, and 1000000 returned records and the three distinct record sizes were 15, 50, and 150 fields per record. All experiments concluded that the JFlexBOT methodology was superior to the POJO approach. Specifically, the reduction of record size was 43%, 49%, and 69% for small, medium, and large records respectively. Additional metrics showed 65% improvement during the read serialization process, and up to 99% improvement during the write serialization time. The total request time latency decreased by 54% for the experiment returning 1 million records of 150 fields per record. With reductions in the size of bytes per record and the improvement in serialization time, JFlexBOT displayed superiority to the POJO methodology.

5.2 Future Work

The JFlexBOT approach drastically improves performance for flat relational database tables. We believe there is a need to further develop this approach not just for flat relational database rows, but also for sub tables associated to these flat database tables. With this in mind, the JDatabaseRecord.java class could be wrapped with a JDatabaseTable.java class that includes a collection of JDatabaseRecords representing the rows of the table, and a collection of JDatabaseTables that represent any sub tables. The JDatabaseTable class would then be responsible for writing all rows to the stream using the writeExternal functions of JDatabaseRecord.java, as well as developing specific serialization code for any sub table queries. Furthermore, both client and server specifications would need to be stored using XML descriptors to allow dynamic mapping of JDatabaseTable and JDatabaseRecord.

The experiments completed for this research utilized a currently deployed data center storing high volumes of data. Because of the sensitivity and fault-tolerant nature of the system, the time allotted for the experiments was limited. A specific time was provided to conduct these experiments and during this period, workload on the system was light to moderate. The prearranged time did not enable sufficient metrics to be gathered when workloads of the data center varied from heavy to light. Future work should include additional experiments to help determine how workload will affect the proposed JFlexBOT methodology compared with the traditional POJO methodology.

Lastly, we propose further work to integrate JFlexBOT with well known persistence layer open source technologies such as Hibernate [45] and Castor [49]. The

ORM solutions presently utilize POJOs for relational data mapping and future work would enable JFlexBOT to be utilized with these implementations and drastically improve performance. Our future work will look at these three suggestions and continue to refine the Java flexible binary object transfer technique.

Bibliography

[1] Sun Microsystems, Inc. (2008), "Java Enterprise Edition," Available from http://java.sun.com/javaee.

[2] Chaumette S., Grange P., Metrot, B., and Vigneras, P., "Implementing a High Performance Object Transfer Mechanism over JikesRVM," Laboratoire Bordelais de Recherche en Informatique, Feb. 2004.

[3] Rahimi, S., Wainer, M., Lewis, D., "An Analysis of Java Distributed Computing Communication Trade-offs: Performance and Programming," Southern Illinois University.

[4] World Wide Web Consortium (2006), "Extensible Markup Language (XML) 1.1," Available from http://www.w3.org/TR/2006/REC-xml11-20060816/.

[5] The Apache Software Foundation (2003), "The Apache HTTP Server Project," Available from http://httpd.apache.org.

[6] Philippsen, M., and Haumacher, B., "More Efficient Object Serialization," *Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, Lecture Notes in Computer Science, vol. 1586, pp. 718–732, 1999.

[7] Courtrai, L., Mahéo, Y., and Raimbault, F., "Espresso: A Library For Fast Transfers of Java Objects," *Myrinet User Group Conference*, Lyon, Sept. 2000.

[8] JBoss Group, "JBoss," Available from http://www.jboss.org/index.html.

[9] Sun Microsystems, Inc. (2002), "JDK 6, JDBC Specification," Available from http://java.sun.com/javase/6/docs/technotes/guides/jdbc/.

[10] E. Meijer and W. Schulte, "Unifying Tables, Objects and Documents," *Proc. Declarative Programming in the Context of OO Languages*, 2003.

[11] World Wide Web Consortium (2007), "SOAP Version 1.2," Available from http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[12] Cecchet, E., J. Marguerite, and W. Zwaenepoel (2002) "Performance and Scalability of EJB Applications," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* pp. 246-261, 2002.

[13] Tucker, P., Bennett, J., and Hill, C., "Method of Creating A Tabular Data Stream For Sending Rows Of Data Between Client And Server," U.S. Patent no. 5974416, Oct. 1999.

[14] Google (2008), "Protocol Buffers: Google's Data Interchange Format," Available from http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[15] Google (2008), "Protocol Buffers: Developers Guide," Available from http://code.google.com/apis/protocolbuffers/docs/overview.html.

[16] Hericko, M., Juric, M., Rozmann, I., Eloglavec, S., and Zivkovic, A., "Object Serialization Analysis and Comparison in Java and .NET," vol. 38, issue 8, pp. 44-54, Aug. 2003.

[17] Opyrchal, L., and Prakash, A., "Efficient Object Serialization in Java," *in Proceedings, 19th IEEE International Conference on Distributed Computing Systems Workshops*, Austin, TX, pp. 96-101, June 1999.

[18] Sun Microsystems, Inc. (2008), "2008 JavaOne Conference," Available from http://java.sun.com/javaone/sf/.

[19] Ambler S., (1997), "Mapping Objects To Relational Databases," Available from http://jeffsutherland.com/objwld98/mappingobjects.pdf.

[20] Park, J. G., and Lee, A. H., "Specializing The Java Object Serialization using Partial Evaluation For A Faster RMI," *Eighth International Conference on Parallel and Distributed Systems*, 2001.

[21] The Jikes RVM Project (2007), "JikesRVM," Available from http://jikesrvm.org/.

[22] Pantaleev, A. and Rountev, A., "Identifying Data Transfer Objects in EJB Applications," *Dynamic Analysis, 2007. WODA '07. Fifth International Workshop*, pp. 5-5, May 2007.

[23] Oracle Corporation, "Oracle 10g Database," Available from http://www.oracle.com/technology/products/database/oracle10g/index.html.

[24] Java Community Process (2004), "JSR-000224 Java API for XML-Based Web Services 2.0," Available from http://jcp.org/aboutJava/communityprocess/pfd/jsr224/index.html.

[25] Bourret, R., Bornhovd, C., and Buchmann, A., "A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases" *Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000),* pp. 134, 2000.

[26] Sun Microsystems, Inc. (2008), "Java Servlet Technology," Available from http://java.sun.com/products/servlet/.

[27] Kono, K., and Masuda, T., "Efficient RMI: Dynamic Specialization of Object Serialization," Computing Systems, 2000. Proceedings. 20th International Conference, Taipei, Taiwan, pp. 308-315, April 2000.

[28] World Wide Web Consortium (2003), "Hyper Text Transfer Protocol," Available from http://www.w3.org/Protocols/.

[29] Juric, M., Kezmah, B., Heriko, M., Rozman, I., and Vezocnik, I., "Java RMI, RMI Tunneling and Web Services Comparison and Performance Analysis," *ACM SIGPLAN Notices*, vol. 39, no. 5, May 2004.

[30] Gu, Y., Hong, X., Mazzucco, M., and Grossman, R., "SABUL: A High Performance Data Transfer Protocol," Available from http://www.rgrossman.com/pdf/sabul-hpdtp-11-02.pdf.

[31] JPOX (2008), "Java Persistent Objects," Available from http://www.jpox.org.

[32] Sun Microsystems Inc. (2008), "Java Data Objects," Available from http://java.sun.com/products/jdo/.

[33] Fernández, M., Kadiyska, Y., Suciu, D., Morishima, A., and Tan, W., "SilkRoute: A Framework for Publishing Relational Data in XML," vol. 27, issue 4, pp. 438-493, Dec. 2002.

[34] Keith M., and Schincariol, M., *Pro EJB 3: Java Persistence API*, Apress, 2006, pp.71-110.

[35] Hirano, S., Yasu, Y., and Igarashi, H., "Performance Evaluation of Popular Distributed Object Technologies for Java," *Concurrency: Practice and Experience*, Vo. 10, issue 11-13, pp. 927 – 940, Dec. 1998.

[36] Gray, N., "Comparison of Web Services, Java-RMI, and CORBA Service Implementations," *AWSA*, Melbourne, pp. 52-63, 2004.

[37] Sarinho, V., "Using JPOX to Develop a Persistence API for Generic Objects," *IFIP International Federation for Information Processing*, Springer Boston, 2008, 993-941.

[38] Takase, T., and Tajima, K., "Efficient Web Services Message Exchange by SOAP Bundling Framework," *11th IEEE International Enterprise Distributed Object Computing Conference,* pp. 63, Oct. 2007.

[39] Graham, S., Simeonov, S., Boubez, T., Daniels, G., Davis, D., Nakamura, Y., Neyama, R., *Building Web Services with Java,* 2d ed. Sams, 2005.

[40] Li, Q., and Moon, B., "Distributed Cooperative Apache Web Server," *Proceedings of the 10th International WWW Conference*, pp. 1215-1229, May 2001.

[41] Barroso, L. A., Dean, J., and Hlzle, U., "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22-28, 2003.

[42] He, X., and Yang, Q., "Performance Evaluation of Distributed Web Servers Under Commercial Workload," *Proceedings of the Internet Conference*, 2000.

[43] Oracle Corporation, "TopLink," Available from http://www.oracle.com/technology/products/ias/toplink/index.html.

[44] Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., and Zwaenepoal, W., "A Comparison of Software Architectures for E-Business Applications," *Technical Report TR02-392*, Rice University, Feb. 2002.

[45] Hibernate (2006), "Hibernate," Available from http://www.hibernate.org/.

[46] Jusic, S., and Peck, L., "PersistF: A Transparent Persistence Framework with Architecture Applying Design Patterns," *Issues in Informing Science and Information Technology*, vol. 4, pp. 767-779, 2007.

[47] White, S., and Graham, J., "System and Method for Improved Serialization of Java Objects," U.S. Patent no 6438559, Aug. 2002.

[48] O'Neil, E., "Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM)," *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, pp. 1351–1356, 2008.

[49] The Castor Project (2005), "Castor" Available from http://www.castor.org/.

[50] Courtrai, L., Maheo, Y., Raimbault, F., "Java Objects Communication on a High Performance Network," Parrallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro, Mantova, Italy, pp. 203-210, February 2001.

[51] Nagappan, R., Skoczylas, R., and Sriganesh, R., *Developing Java Web Services*, Wiley Computer Publishing, 2003.

[52] Sun Microsystems Inc. (2007), "Java Serialization API," Available from http://java.sun.com/javase/6/docs/technotes/guides/serialization/index.html.

[53] Breg, F., Polychronopoulos, C., "Java virtual machine support for object serialization," *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, Palo Alto, California, 2001.

[54] Breg F, Gannon DB, "A customizable implementation of RMI for high performance computing," *Parallel and Distributed Processing, 11 IPPS/SPDP'99 Workshops*

Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, pp. 733–747, 1999.

[55] Carpenter, B., Fox, G., Ko, S., and Lim, S., "Object serialization for marshaling data a Java interface to MPI," *Porceedings of the ACM 1999 conference on Java Grande*, San Francisco, California, 1999.

[56] Agile Data (2006), "The Object-Relational Impedance Mismatch," Available from http://www.agiledata.org/essays/impedanceMismatch.html.

[57] Ersoz, D., "Exploring User-Level Communication in Designing Cluster-Based Data Centers", The Pennsylvania State University, 2007.

[58] Sun Microsystems, Inc. (2008), "Application Server Diagram," Available from http:// docs.sun.com/app/docs/doc/819-3671/ablat?a=view.

Appendix A

JDatabaseRecord.java Source Code

```
package com.rsc.asap.common.services;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.nio.ByteBuffer;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import com.rsc.asap.common.enums.ColumnType;

public class JDatabaseRecord implements Externalizable {

    /** Data arrays used to store the data of the database record. **/
    transient public boolean[] booleanData;
    transient public char[] charData;
    transient public byte[] byteData;
    transient public short[] shortData;
    transient public int[] intData;
    transient public long[] longData;
    transient public float[] floatData;
    transient public double[] doubleData;
    transient public String[] stringData;

    /** Arrays that will store the null and zero values**/
    transient public boolean[] nullData;
    transient public boolean[] zeroData;

    /** Hashmap of each column to the specific index into the paricular
        data array. This transient mapping should be set up by the
        JDatabaseTable class. **/
    transient public HashMap<Integer, Integer> dataMapping;

    // serialVersionUID
    private static final long serialVersionUID = 13L;

    // Variable used for client side column counts
    private static short[] staticCounters;
    private static boolean readCounters;

    // Used in the readExternal function to store current records counts
    transient private short[] counters;

    // Global index used to index into the null and zero vectors.
    transient private short arrayIndex;

    // Buffer used when reading data from ResultSet.
    transient private ByteBuffer byteBuffer;

    /**
```

```
 * Default constructor that will be used on the client machine when
 * the serialized object is de-serialized. This constructor reads
 * system properties the total number of columns to read, and count
 * each type of data source from system properties.
 * ©since 4/27/08
 */
public JDatabaseRecord() {

    if (readCounters) {

        // Read in system properties for the number of columns
        staticCounters = new short[ColumnType.values().length];
        StringBuffer counterString = new StringBuffer("count");

        for (int i = 0; i < staticCounters.length; i++) {

            counterString.append(i);
            staticCounters[i] = Integer.valueOf(System.getProperty(
                counterString.toString(), "0")).shortValue() ;
            counterString.delete(counterString.length() - 1,
                counterString.length());
        }

        // Set the flag so the counters don't have to be read in.
        readCounters = false;
    }

    counters = new short[ColumnType.values().length];
    for (int i = 0; i < counters.length; i++) {

        counters[i] = staticCounters[i];
    }
}

/** Empty constructor that is used on the server side to create
 * a new JDatabaseRecord without obtaining the system properties.
 *
 * @param setupCounts - Parameter differentiates default constructor
 */
public JDatabaseRecord(boolean noProperties) {

    counters = new short[ColumnType.values().length];
}

/**
 * Function used to set up the data to be transfered based on
 * the result set and columns of the result set.
 * @param results - The result set to obtain the data from.
 * @param columnMapping - A hashmap that contains lists of column
 *                        indices used for each type of ColumnType.
 *                        This allows the data from the ResultSet to
 *                        be stored in primitive arrays.
 * @param numColumns - The total number of columns to set.
 */
public void setValues(ResultSet results,
   HashMap<ColumnType, ArrayList<Integer>> columnMapping,
   int numColumns) {

    try {

        ArrayList<Integer> columns;
        int nullIndex = 0, zeroIndex = 0;
```

```
byteBuffer = ByteBuffer.allocate(1024);

// Set up the null and zero arrays.
nullData = new boolean[numColumns] ;
zeroData = new boolean[numColumns];

ColumnType[] columnTypes = ColumnType.values();
for (int key = 1; key < columnTypes.length; key++) {

    // Obtain a list of indices based on the columns
    // so that the data can be read for each type and written
    // to the stream accordingly.
    columns = columnMapping.get(columnTypes[key]);

    // NOTE: It is important that the list that is returned
    // for each type of the mapping is never null. If there
    // are no columns of the specific type, then an empty
    // list should be created so that the particular array of
    // primitive data that is written to the stream during
    // serialization will not throw a null pointer exception
    // for not having a length. Based on the type of data, go
    // through each column and write the column. Here the
    // enumeration for the different types are used, instead
    // of the SQL mapping.This is because the hashmap that is
    // a parameter should have already taken the select
    // columns and determined the column type.
    switch (columnTypes[key]) {

        case BOOLEAN_COLUMN:

            booleanData = new boolean[columns.size()];
            for (int i = 0; i < columns.size(); i++,
               nullIndex++, zeroIndex++) {

                booleanData[counters[
                    ColumnType.BOOLEAN_COLUMN.ordinal()]] =
                    results.getBoolean(columns.get(i));

                if (results.wasNull()) {
                    nullData[nullIndex] = true;
                } else {

                     counters[ColumnType.BOOLEAN_COLUMN.
                            ordinal()]++;
                }
            }
            break;

        case CHAR_COLUMN:

            String tempChar;
            for (int i = 0; i < columns.size(); i++,
               nullIndex++, zeroIndex++) {

                try {

                    tempChar = results.getString(
                     columns.get(i));
                    if (tempChar == null) {
                        nullData[nullIndex] = true;
                    } else {
                        byteBuffer.putChar(
```

```
            tempChar.charAt(0)) ;
           }

        } catch (IndexOutOfBoundsException ie) {

            nullData[nullIndex] = true;
        }
    }

    break;

case BYTE_COLUMN:

    byte byteData;
    for (int i = 0; i < columns.size(); i++,
       nullIndex++, zeroIndex++) {

        byteData = results.getByte(columns.get(i));
        if (results.wasNull()) {
            nullData[nullIndex] = true;
        } else if (byteData == 0) {
            zeroData[zeroIndex] = true;
        } else {
            byteBuffer.put(byteData);
        }
    }
    break;

case SHORT_COLUMN:

    short shortData;
    for (int i = 0; i < columns.size(); i++,
       nullIndex++, zeroIndex++) {
        shortData = results.getShort(columns.get(i));
        if (results.wasNull()) {
            nullData[nullIndex] = true;
        } else if (shortData == 0) {
            zeroData[zeroIndex] = true;
        } else {
            byteBuffer.putShort(shortData);
        }
    }
    break;

case INT_COLUMN:

    int intData;
    for (int i = 0; i < columns.size(); i++,
       nullIndex++, zeroIndex++) {
        intData = results.getInt(columns.get(i));
        if (results.wasNull()) {
            nullData[nullIndex] = true;
        } else if (intData == 0) {
            zeroData[zeroIndex] = true;
        } else {
            byteBuffer.putInt(intData);
        }
    }

    break;
case LONG_COLUMN:
```

```
    long longData;
    for (int i = 0; i < columns.size(); i++,
       nullIndex++, zeroIndex++) {
        try {
            longData = results.getLong(
             columns.get(i));
        } catch (SQLException date) {
            try {
                longData = results.getDate(
                    columns.get(i)).getTime();
            } catch (SQLException timestamp) {
                longData = results.getTimestamp(
                    columns.get(i)).getTime();
            }
            if (results.wasNull()) {
                nullData[nullIndex] = true;
            } else if (longData == 0) {
                zeroData[zeroIndex] = true;
            } else {
                byteBuffer.putLong(longData);
            }
        }
    }
    break;

case FLOAT_COLUMN:

    float floatData;
    for (int i = 0; i < columns.size(); i++,
       nullIndex++, zeroIndex++) {
        floatData = results.getFloat(columns.get(i));
        if (results.wasNull()) {
            nullData[nullIndex] = true;
        } else if (floatData ==0) {
            zeroData[zeroIndex] = true;
        } else {
            byteBuffer.putFloat(floatData) ;
        }
    }
    break;

case DOUBLE_COLUMN:

    double doubleData;
    for (int i = 0; i < columns.size(); i++,
       nullIndex++, zeroIndex++) {
        doubleData = results.getDouble(
             columns.get(i));
        if (results.wasNull()) {
            nullData[nullIndex] = true;
        } else if (doubleData == 0) {
            zeroData[zeroIndex] = true;
        } else {
            byteBuffer.putDouble(doubleData) ;
        }
    }

    break;

case STRING_COLUMN:

    String tempString;
```

```
                    for (int i = 0; i < columns.size(); i++,
                       nullIndex++, zeroIndex++) {
                        tempString = results.getString(
                              columns.get(i));
                        if (tempString == null) {
                            nullData[nullIndex] = true;
                        } else {

                        }
                        byteBuffer.putShort(
                              (short)tempString.length());
                        byteBuffer.put(tempString.getBytes());
                    }

                    break;
            }
        }

    // Go through the data and set up the null, zero, and arrays.
    // Catch any exceptions
    } catch (SQLException sql) {

        sql.printStackTrace() ;
        System.err.println(sql.getMessage()) ;
    }
}
/**
 * Function used to set up the boolean flag that states a new
 * counter is needed to be read.
 * @param readCounters - If new counters need to be read.
 */
public static void resetColumnCounter(boolean counter) {

    readCounters = counter;
}

/**
 * This function should be called from the JDatabaseTable class on
 * the client machine to set up the data mapping of this record. This
 * function will set up the mapping of data of each type of the
 * array, where columns are represented by integers (i.e. the first
 * column is 0, second column is represented by 1, etc...I. NOTE:
 * This function should be called after the object is read from the
 * stream to ensure that the counters are configured correctly.
 *
 * ©since 4/27/08
 */
public void configureDataMapping() {

    if (dataMapping == null) {

          dataMapping = new HashMap<Integer, Integer>();
    }

    else {

          dataMapping.clear();
    }

    // Set up the index to be zero.
    int globalIndex = 0;
    int specificIndex = 0;
```

```
    for (int i = 0; i < counters.length; i++) {

        specificIndex = 0;
        while (specificIndex < counters[i]) {

            if (!nullData[globalIndex] && !zeroData[globalIndex]) {

                dataMapping.put(globalIndex, specificIndex);
                globalIndex++;
                specificIndex++;
            }
        }
    }
}

/*
 * (non-Javadoc)
 *
 * ©see Java.io.ExternalizableftreadExternal(Java.io.ObjectInput)
 */
public void readExternal(Objectlnput in) throws lOException {

    // Create the null and zero data arrays of this length;
    nullData = new boolean[counters[
          ColumnType.COLUMN_COUNT.ordinal()]];
    zeroData = new boolean[counters[
          ColumnType.COLUMN_COUNT.ordinal()]];

    // Based on the number of columns (which will be known to both
    // the client and server), configure and output the the number of
    // bits that are in the nullData and zeroData boolean arrays.
    // Determein how many bits there should be, then read the next
    // number of bytes from the stream where 1 byte = 8 bits, 8
    // numDataColumns is the number of bytes to read.
    arrayIndex = 0 ;
    while (arrayIndex < nullData.length) {

        // Read in the byte
        byte b = in.readByte();

        // Possibly need to shift the byte if not all bits in the
        // byte are used for the array.
        int shift = 8 - (nullData.length - arrayIndex);
        b <<= (shift > 0) ? shift : 0;
        for (int j = 0; j < 8 && arrayIndex < nullData. length;
          j++, arrayIndex++) {

            if ((b & 0x80) == 0x80) nullData[arrayIndex] = true;
            else nullData[arrayIndex] = false;
            b <<= 1;
        }
    }

    arrayIndex = 0;
    while (arrayIndex < zeroData.length) {

        // Read in the byte
        byte b = in.readByte() ;

        // Possibly need to shift the byte if not all bits in the
        // byte are used for the array.
        int shift = 8 - (zeroData.length - arrayIndex);
```

```
    b <<= (shift > 0) ? shift : 0;
    for (int j = 0; j < 8 && arrayIndex < zeroData.length;
         j++, arrayIndex++) {

        if ((b & 0x80) == 0x80) zeroData[arrayIndex] = true;
        else zeroData[arrayIndex] = false;
        b <<= 1;
    }
}

// Now that the data is ready to be read in, start the global
// array index to be zero.
arrayIndex = 0;

// Next, obtain the data to be transfered following the below
// algorithm
int booleanIndex = 0;
booleanData = new boolean[getArraySize(
      counters[ColumnType.BOOLEAN_COLUMN.ordinal()])];
while (booleanIndex < booleanData.length) {

    // Read in the byte
    byte b = in.readByte() ;

    // Possibly need to shift the byte if not all bits in the
    // byte are used for the boolean array.
    int shift = 8 - (booleanData.length - booleanIndex);
    b <<= (shift > 0) ? shift : 0 ;
    for (int j = 0; j < 8 && booleanIndex < booleanData.length;
      j++, booleanIndex++) {

        if ((b & 0x80) == 0x80) booleanData[booleanIndex] = true;
        else booleanData[booleanIndex] = false;
        b <<= 1;
    }
}

// 2. char values
charData = new char[getArraySize(
      counters[ColumnType.CHAR_COLUMN.ordinal()])];
for (int i = 0; i < charData.length; i++) {
    charData[i] = in.readChar() ;
}

// 3. byte values
byteData = new byte[getArraySize(
      counters[ColumnType.BYTE_COLUMN.ordinal()])];
for (int i = 0; i < byteData.length; i++) {

    byteData[i] = in.readByte();
}

// 4. short values
shortData = new short[getArraySize(
      counters[ColumnType.SHORT_COLUMN.ordinal()])];
for (int i = 0; i < shortData.length; i++) {
    shortData[i] = in.readShort();
}

// 5. int values
intData = new int[getArraySize(
      counters[ColumnType.INT_COLUMN.ordinal()])];
```

```
        for (int i = 0; i < intData.length; i++) {
            intData[i] = in.readint() ;
        }

        // 6. long values
        longData = new long[getArraySize(
              counters[ColumnType.LONG_COLUMN.ordinal()])];
        for (int i = 0; i < longData.length; i++) {
            longData[i] = in.readLong();
        }

        // 7. float values
        floatData = new float[getArraySize(
              counters[ColumnType.FLOAT_COLUMN.ordinal()])];
        for (int i = 0; i < floatData.length; i++) {
            floatData[i] = in.readFloat() ;
        }

        // 8. double values
        doubleData = new double[getArraySize(
              counters[ColumnType.DOUBLE_COLUMN.ordinal()])];
        for (int i = 0; i < doubleData.length; i++) {
            doubleData[i] = in.readDouble() ;
        }

        // 9. String values
        stringData = new String[getArraySize(
              counters[ColumnType.STRING_COLUMN.ordinal()])];
        short stringLength;
        byte[] val;
        for (int i = 0; i < stringData.length; i++) {

            stringLength = in.readShort();
            val = new byte[stringLength];
            for (int j = 0; j < stringLength; j++) {
                val[j] = in.readByte();
            }

            // Don't need to worry about the val array being of size
            // less than zero because the writeExternal would have never
            // written the string out.
            stringData[i] = new String(val);
        }
    }

/* (non-Javadoc)
 * @see java.io.ExternalizableftwriteExternal(java.io.ObjectOutput)
 */
    public void writeExternal(ObjectOutput out) throws IOException {

        // Compute the number of bits to send for the null and zero data.
        // Place them into a byte for each 8 columns. Then write out this
        // byte. Do this for both the null array and the zero array.
        arrayIndex = 0 ;
        while (arrayIndex < nullData.length) {

            // Loop through the 8 bits based on the nullData array.
            byte b = 0;
            for (int j = 0; j < 8 && arrayIndex < nullData.length;
              j++, arrayIndex++) {

                // Shift the bits to the left one by one, and set up the
```

```
                // bit to be true or false based on the null data.
                b <<= 1;

                // If null, then logical OR the specific bit with a mask
                // to place a 1 into the specific bit location.
                if (nullData[arrayIndex]) b |= 1;
            }

            // Send the byte to the stream.
            out.writeByte(b);
        }

        arrayIndex = 0;
        while (arrayIndex < zeroData.length) {

            // Loop through the 8 bits based on the nullData array.
            byte b = 0;
            for (int j = 0; j < 8 && arrayIndex < zeroData.length;
              j++, arrayIndex++) {

                // Shift the bits to the left one by one, and set up the
                // bit to be true or false based on the null data.
                b <<= 1;

                // If null, then logical OR the specific bit with a mask
                // to place a 1 into the specific bit location.
                if (zeroData[arrayIndex]) b |= 1;
            }

            // Send the byte to the stream.
            out.writeByte(b) ;
        }

        arrayIndex = 0;
        while (arrayIndex <
counters[ColumnType.BOOLEAN_COLUMN.ordinal()]) {

            // Loop through the 8 bits based on the nullData array.
            byte b = 0;
            for (int j = 0; j < 8 && arrayIndex <
                    counters[ColumnType.
                    BOOLEAN_COLUMN.ordinal()]; j++, arrayIndex++) {

                // Shift the bits to the left one by one, and set up the
                // bit to be true or false based on the null data.
                b <<= 1;

                // If null, then logical OR the specific bit with a mask
                // to place a 1 into the specific bit location.
                if (booleanData[arrayIndex]) b |= 1;
            }

            // Send the byte to the stream.
            out.writeByte(b) ;
        }

        // Now simply write out the byte array
        byteBuffer.flip() ;
        byte[] copy = new byte[byteBuffer.limit()] ;
        byteBuffer.get(copy, 0, byteBuffer.limit());
        out.write(copy) ;
    }
```

```
    // Helper function used to return the size of the array that should
    // be created based on the number of possible columns. This function
    // checks against the null and zero arrays and determines what the
    // size of the array should be. The global counter, arraylndex will
    // be incremented for every numOfColumn to check null and zero
    //
    // @param count - The expected total count of columns for the type.
    // ©return short - The size of the array that is to be created.
    private short getArraySize(short count) {

        // Initialize the arraySize variable to be null and increment the
        // size for every non null and zero value.
        short arraySize = 0;
        while (count > 0) {

            if (!nullData[arrayIndex] && !zeroData[arrayIndex]) {
                arraySize++;
            }

            // Increment the global array index counter, decrement
            // the number that the array should be.
            arrayIndex++;
            count-- ;
        }
        return arraySize;
    }
}
```

Appendix B

ColumnTypeEnum.java Source Code

```
package com.rsc.asap.common.enums;

public enum ColumnType {

    COLUMN_COUNT,
    BOOLEAN_COLUMN,
    CHAR_COLUMN,
    BYTE_COLUMN,
    SHORT_COLUMN,
    INT_COLUMN,
    LONG_COLUMN,
    FLOAT_COLUMN,
    DOUBLE_COLUMN,
    STRING_COLUMN;
}
```

Appendix C

Java / SQL Mapping Source Code

```
// Helper function that will take in a result set and
// create a hashmap of the column types and the result
// set columns in a list.
//
// @param rs - The result Set
// @param columnTypes - The hashmap of column lists.
private int mapColumnTypes(ResultSet rs, HashMap<ColumnType,
ArrayList<Integer>>        columnTypes) {

    int totalColumns = 0;

    try {

        ResultSetMetaData metaData = rs.getMetaData();

        // Create lists for each type.
        ColumnType[] types = ColumnType.values();
        ArrayList<ArrayList<Integer>> lists = new
             ArrayList<ArrayList<Integer>>();
        for (int i=0; i < types.length; i++) {

            lists.add(new ArrayList<Integer>());
        }

        totalColumns = metaData.getColumnCount();

        //Go through adding the column numbers to the lists for each type.
        int precision, scale, unknownCounter = 0;
        for (int i = 1; i <= totalColumns; i++) (

                switch (metaData.getColumnType(i)) {

                    case Types.BOOLEAN:

                        lists.get(ColumnType.BOOLEAN_COLUMN.ordinal()).add(i);
                        break;

                    case Types.CHAR:

                        lists.get(ColumnType.CHAR_COLUMN.ordinal()).add(i);
                        break;

                    case Types.TINYINT:

                        lists.get(ColumnType.BYTE_COLUMN.ordinal()).add(i);
                        break;

                    case Types.SMALLINT:

                        lists.get(ColumnType.SHORT_COLUMN.ordinal()).add(i);
                        break;
```

```
case Types.NUMERIC:

    precision = metaData.getPrecision(i);
    scale = metaData.getScale(i);

    // The number is a float or double value.
    if (scale > 0) {

        // Set up the number to be a float value.
        if (scale <= 6) {

              lists.get(ColumnType.FLOAT_COLUMN.
              ordinal()).add(i);
        }

        // Set the number up to be a double value.
        else {

              lists.get(ColumnType.DOUBLE_COLUMN.
              ordinal()).add(i);
        }
    }

    // The number is a byte, short, int, or long value
    else {

        // Set up the byte value.
        if (precision <= 2) (

              lists.get(ColumnType.BYTE_COLUMN.
              ordinal()).add(i);
        }

        else if (precision <=4) {

              lists.get(ColumnType.SHORT_COLUMN.
              ordinal()).add(i);
        }

        else if (precision <= 9) {

              lists.get(ColumnType.INT_COLUMN.
              ordinal()).add(i);
        }

        else {

              lists.get(ColumnType.LONG_COLUMN.
              ordinal()).add(i) ;
        }
    }
    break;

case Types.INTEGER:

    lists.get(ColumnType.INT_COLUMN.ordinal()).add(i);
    break;

case Types.DATE:
case Types.TIMESTAMP:
case Types.BIGINT:
```

```
                        lists.get(ColumnType.LONG_COLUMN.ordinal()).add(i);
                        break;

                    case Types.DECIMAL:
                    case Types.REAL:
                    case Types.FLOAT:

                        lists.get(ColumnType.FLOAT_COLUMN.ordinal()).add(i);
                        break;

                    case Types.DOUBLE:

                        lists.get(ColumnType.DOUBLE_COLUMN.ordinal()).add(i);
                        break;

                    case Types.NVARCHAR:
                    case Types.LONGNVARCHAR:
                    case Types.LONGVARCHAR:
                    case Types.VARCHAR:

                        lists.get(ColumnType.STRING_COLUMN.ordinal()).add(i);
                        break;

                    default:

                        unknownCounter++;
                        break;
                }
        }

        // Set up the hashmap that holds the lists of indices
        // per columnType.
        for (int i = 1; i < types.length; i++) {

            columnTypes.put(types[i], lists.get(i));
        }

    } catch (SQLException sql) (

            sql.printStackTrace();
            System.err.print In(sql.getMessage());
    }

    return totalColumns;
}
```

Appendix D

POJO Experiment Metrics

15 Fields Per / Record

Experiment	Read Serialize (msec)	Write Serialize (msec)	Server Process (msec)	Total Size (mb)	Bytes per Record	Transfer Time (msec)	Throughput (recs/sec)	Total Request Time (sec)
25000								
Run 1	6848	7035	919	2.701	108.04	7578	3299.02	19.40
Run 2	6637	6832	655	2.701	108.04	7065	3538.57	17.06
Run 3	8254	8371	591	2.701	108.04	8663	2885.84	18.78
100000								
Run 1	28069	28582	2541	10.8	108.02	29407	3400.55	49.25
Run 2	28829	29284	2456	10.8	108.02	30340	3295.98	50.09
Run 3	25457	27717	2548	10.8	108.02	28794	3472.95	48.53
250000								
Run 1	71791	72233	6566	27.0	108.01	75250	3322.26	113.83
Run 2	69093	68494	6546	27.0	108.01	72356	3455.19	115.00
Run 3	68323	68795	6417	27.0	108.01	71762	3483.74	113.17
1000000								
Run 1	303718	302398	30431	108.0	108.01	322103	3104.60	472.22
Run 2	289661	286776	23080	108.0	108.01	303365	3296.36	436.81
Run 3	288699	289851	27315	108.0	108.01	302898	3301.44	456.45

50 Fields Per / Record

Experiment	Read Serialize (msec)	Write Serialize (msec)	Server Process (msec)	Total Size (mb)	Bytes per Record	Transfer Time (msec)	Throughput (recs/sec)	Total Request Time (sec)
25000								
Run 1	20907	20786	2187	7.48	299.16	21603	1157.25	35.46
Run 2	18422	18605	1984	7.48	299.16	19076	1310.55	34.02
Run 3	18899	18928	2064	7.48	299.16	19591	1276.10	32.33
100000								
Run 1	72358	72257	12036	29.86	298.63	74463	1342.95	110.88
Run 2	75983	75849	8729	29.86	298.63	78522	1273.53	111.39
Run 3	77734	78281	10222	29.86	298.63	80731	1238.68	114.88
250000								
Run 1	199872	200138	21235	74.61	298.43	205924	1214.04	277.11
Run 2	184553	184516	21630	74.61	298.43	191012	1308.82	260.55
Run 3	187159	187216	20272	74.61	298.43	193308	1293.27	269.41
1000000								
Run 1	927411	901871	169181	298.37	298.37	953220	1210.24	1552.44
Run 2	877389	847332	355114	298.37	298.37	901809	1299.50	1789.20
Run 3	905614	888242	198937	298.37	298.37	932157	1298.77	1410.16

150 Fields Per / Record

Experiment	Read Serialize (msec)	Write Serialize (msec)	Server Process (msec)	Total Size (mb)	Bytes per Record	Transfer Time (msec)	Throughput (recs/sec)	Total Request Time (sec)
25000								
Run 1	53713	53914	9318	22.25	890.01	54636	457.57	85.81
Run 2	54749	54898	8959	22.25	890.01	55604	449.61	77.94
Run 3	58853	58668	8074	22.25	890.01	59425	420.70	81.39
100000								
Run 1	206320	207071	37250	88.70	887.04	209497	477.33	287.35
Run 2	215224	215267	36765	88.70	887.04	218097	458.51	289.96
Run 3	214808	216241	39705	88.70	887.04	218703	457.24	298.13
250000								
Run 1	511682	511447	118740	221.43	886.28	518592	482.33	749.00
Run 2	530503	531837	130425	221.43	886.28	539870	463.07	829.82
Run 3	505926	505047	136056	221.43	886.28	512352	487.95	752.65
1000000								
Run 1	2224333	2196667	521293	886.19	886.19	2254507	443.56	3178.89
Run 2	2284736	2291243	572333	886.19	886.19	2317714	431.50	3227.18
Run 3	2089326	2095814	475407	886.19	886.19	2120588	471.57	3002.58

Appendix E

JFlexBOT Experiment Metrics

15 Fields Per / Record

Experiment	Read Serialize (msec)	Write Serialize (msec)	Server Process (msec)	Total Size (mb)	Bytes per Record	Transfer Time (msec)	Throughput (recs/sec)	Total Request Time (sec)
25000								
Run 1	3722	58	771	1.54	61.41	4383	5703.86	5.70
Run 2	3524	68	655	1.54	61.41	4319	5778.38	5.79
Run 3	3463	64	706	1.54	61.41	4138	6041.57	6.04
100000								
Run 1	13370	266	2641	6.15	61.51	15961	6265.27	35.23
Run 2	13919	259	3024	6.15	61.51	16350	6116.21	37.96
Run 3	13590	249	2914	6.15	61.51	15894	6291.68	42.52
250000								
Run 1	34954	836	7788	15.39	61.54	40950	6105.01	84.52
Run 2	34477	641	6595	15.39	61.54	40438	6182.30	83.60
Run 3	38058	617	6596	15.39	61.54	43757	5713.37	83.40
1000000								
Run 1	138777	2561	36764	61.56	61.56	163240	6125.95	330.34
Run 2	154896	2426	28106	61.56	61.56	186228	5369.76	336.43
Run 3	156430	2527	26972	61.56	61.56	179815	5561.27	323.05

50 Fields Per / Record

Experiment	Read Serialize (msec)	Write Serialize (msec)	Server Process (msec)	Total Size (mb)	Bytes per Record	Transfer Time (msec)	Throughput (recs/sec)	Total Request Time (sec)
25000								
Run 1	9001	82	1557	3.85	153.85	9896	2526.27	2.53
Run 2	9421	79	1377	3.85	153.85	10001	2499.75	2.50
Run 3	13367	76	1377	3.85	153.85	14132	1769.03	1.77
100000								
Run 1	39455	309	5496	15.37	153.67	42046	2378.35	70.13
Run 2	39946	288	6353	15.37	153.67	42452	2355.60	71.10
Run 3	39034	294	6253	15.37	153.67	41518	2408.59	72.91
250000								
Run 1	96112	796	15361	38.46	153.83	108791	2297.98	170.66
Run 2	95234	791	16644	38.46	153.83	101927	2452.74	173.39
Run 3	93663	757	15778	38.46	153.83	100079	2498.03	169.11
1000000								
Run 1	640805	3294	293095	153.32	153.32	687007	1746.71	1397.05
Run 2	672938	3007	366167	153.32	153.32	713163	1702.18	1512.62
Run 3	604073	3467	170720	153.32	153.32	646450	1872.77	1161.55

150 Fields Per / Record

Experiment	Read Serialize (msec)	Write Serialize (msec)	Server Process (msec)	Total Size (mb)	Bytes per Record	Transfer Time (msec)	Throughput (recs/sec)	Total Request Time (sec)
25000								
Run 1	18450	130	3808	6.97	278.75	19106	1308.49	39.62
Run 2	17166	122	2878	6.97	278.75	18004	1388.58	35.95
Run 3	18982	99	4432	6.97	278.75	19672	1270.84	36.67
100000								
Run 1	71514	463	18910	27.3	272.97	74319	1345.55	126.88
Run 2	68175	425	16133	27.3	272.97	70963	1409.19	131.46
Run 3	70681	1033	19639	27.3	272.97	72682	1375.86	135.75
250000								
Run 1	183485	1752	67269	68.13	272.53	188864	1323.70	346.68
Run 2	172789	1878	88665	68.13	272.53	178563	1400.07	369.19
Run 3	200665	1287	67861	68.13	272.53	207181	1206.67	366.68
1000000								
Run 1	733940	7008	269076	272.45	272.45	755456	1356.70	1375.80
Run 2	691156	7512	354660	272.45	272.45	714252	1403.36	1465.21
Run 3	802660	5418	271444	272.45	272.45	828724	1303.12	1340.78