DJ: Bridging Java and Deductive Databases

Andrew B. Hall

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Dr. Eli Tilevich, Chair
Dr. Osman Balci
Dr. Naren Ramakrishnan

May 30, 2008
Blacksburg, Virginia

Keywords and phrases: Database Management System (DBMS), Database, Deductive Database, Column Oriented Database, Object Oriented (OO) Language, Orthogonal Persistence, Plain Old Java Object (POJO), Plain Old Java Interface (POJI), Dynamic Proxy, Middleware

DJ: Bridging Java and Deductive Databases

Andrew B. Hall

ABSTRACT

Modern society is intrinsically dependent on the ability to manage data effectively. While relational databases have been the industry standard for the past quarter century, recent growth in data volumes and complexity requires novel data management solutions. These trends revitalized the interest in deductive databases and highlighted the need for column-oriented data storage. However, programming technologies for enterprise computing were designed for the relational data management model (i.e., row-oriented data storage). Therefore, developers cannot easily incorporate emerging data management solutions into enterprise systems.

To address the problem above, this thesis presents Deductive Java (DJ), a system that enables enterprise programmers to use a column oriented deductive database in their Java applications. DJ does so without requiring that the programmer become proficient in deductive databases and their non-standardized, vendor-specific APIs. The design of DJ incorporates three novel features: (1) tailoring orthogonal persistence technology to the needs of a deductive database with column-oriented storage; (2) using Java interfaces as a primary mapping construct, thereby simplifying method call interception; (3) providing facilities to deploy light-weight business rules.

DJ was developed in partnership with LogicBlox Inc., an Atlanta based technology startup.

Acknowledgments

I consider my graduate school experience at Virginia Tech to have been an invaluable learning experience in preparing me for the rest of my life. No valuable experience or accomplishment is ever achieved alone or without the help, love, and support of many others. I would like to take this opportunity to thank those who have been most influential and supportive of my graduate studies.

First and foremost I must thank my family, especially my parents and sister. Not only did they encourage me to return to school, but we had many frequent conversations in which they offered great advice and encouragement along the way. I know their frequent prayers and encouraging words aided greatly in the completion of my degrees.

Second of all I would like to thank my adviser Dr. Eli Tilevich, and the other two members of my committee: Dr. Osman Balci and Dr. Naren Ramakrishnan. Dr. Tilevich was an invaluable resource in my academic endeavors through graduate school. He was able to coordinate my graduate research assistantship through connecting me with LogicBlox, and was always there to discuss anything with (especially late at night). I greatly enjoyed the late night coffee runs, and two trips that we made to Atlanta together to meet with LogicBlox. He also taught me how to approach a project from a research perspective, and how to turn the results of a project into a research publication.

Dr. Balci provided invaluable insights into the presentation, and writeup of the project. He also taught me two classes as an undergraduate student, including software engineering, and provided a letter of recommendation when I was applying to graduate school. Dr.

Ramakrishnan has always been incredibly welcoming whenever I have had the occasion to spend time with him, and I greatly enjoyed his "Data Mining" class.

Next I would like to thank the kind folks at LogicBlox. The work presented in this thesis was the result of a partnership with the company. Wes Hunter and Dave Zook spent many hours helping us to grasp the concepts, challenges, and an understanding of the LogicBlox technology. Molham Aref believed in the project enough to fund us, and there were many other people at LogicBlox who contributed technical insights such as Greg Brooks, Soeren Oleson, Mark Bloemeke, and Steve Coulson. Special thanks go Erin Hunter, who handled all of the financial arrangements both with Virginia Tech, and for traveling to Atlanta.

I am incredibly thankful for all of my friends and roommates who were so supportive over the past two years. They were always there to say a kind word, provide encouragement, and in some cases even look over my thesis. My roommates put up with my hectic schedule, and even managed to not see me for days at a time. I would also like to thank my graduate school partners and lab-mates who I worked with here at Virginia Tech, especially Wes Tansey, who was always there to talk over things, and evaluate whether they made sense from an objective point of view.

Finally I would like to thank my future employer Microsoft Corporation for the opportunity to apply the lessons I have learned as a student at Virginia Tech solving real world problems.

Contents

Abstract ii

Acknowledgments iii

List of Figures viii

List of Tables ix

List of Acronyms x

1 Introduction 1

1.1 Motivation For Change 2

1.2 Literature Review of Deductive Databases 4

1.2.1 Database Evolution 6

1.2.2 Brief History of Deductive Databases 7

1.3 Statement of the Problem 10

1.4 Statement of Objectives 11

1.5 Overview of Thesis 11

2 LogicBlox: A Real World Need 13

2.1 LogicBlox Implementation Overview 13

2.2 Overview of Column-Oriented Databases 15

2.3 Objectives 19

2.4 Challenges 22

3 Related Work **25**

3.1 Database Connectivity 26

3.1.1 Users 29

3.1.2 Application Developers 30

3.1.3 DBMS Vendors 32

3.1.4 Connectivity Implementations 34

3.2 Object Oriented Data Mapping 39

3.2.1 Orthogonal Persistence 40

3.2.2 Microsoft LINQ 43

3.3 Applicability of Existing Technologies 45

3.3.1 JDBC 45

3.3.2 Java Hibernate 46

3.3.3 Microsoft LINQ 46

4 DJ: A Java to Deductive Database Bridge **48**

4.1 Solution Approaches 48

4.1.1 JDBC Like Approach 49

4.1.2 Meta Data Types 49

4.1.3 Meta-Data Datalog Queries 50

4.1.4 Database Representative Java Classes 50

4.1.5 Classes versus Interfaces 51

4.1.6 Meta-Data Bindings 51

4.2 DJ Implementation 52

4.2.1 Example in Datalog 54

4.2.2 Basic “Bind” Functionality 55

4.2.3 More Advanced “Bind” Functionality 57

4.2.4 Getter – Setter Functionality 58

4.2.5 Set Handling 60

4.2.6 Business Rule Support 61

4.2.7 Using the Annotated Interfaces 63
4.2.8 Generating Annotated Interfaces 64
4.3 Evaluation 65
4.3.1 Annotated Approach 68
4.3.2 Code Generation 70

5 Conclusions and Future Research **72**
5.1 Conclusions 72
5.2 Contributions 74
5.2.1 Introduction of POJIs 74
5.2.2 Object Oriented to Column Oriented Mapping 75
5.2.3 Automatic deployment of business rules 76
5.3 Future Research 77
5.3.1 Dynamic Queries 77
5.3.2 Asynchronous Queries 78
5.3.3 Advanced Business Rules 78
5.3.4 Transaction Management 78
5.3.5 IDE Support 79
5.3.6 Query Batching 79
5.3.7 Multiple Data Sources 79

Bibliography **81**

List of Figures

2.1 Project Visualization 20
3.1 Database communication through a vendor API 26
3.2 Database communication through a vendor-independent standard 28
3.3 The user's perspective of the need for database connectivity 29
3.4 The application developer's perspective of the need for database connectivity 31
3.5 The DBMS vendor's perspective of the need for database connectivity 33
3.6 ODBC Components 36
3.7 A component DBMS architecture using OLE DB interfaces 37
3.8 JDBC Architecture 38
3.9 Orthogonal Persistence Architecture 42
3.10 Java Hibernate Example 43
3.11 Sample C# DLinq Application 44
4.1 Code Generation Tool 66

List of Tables

1.1 Major RDBMS Releases 6
4.1 Item Database Visualization 53
4.2 LineItem Database Visualization 53
4.3 Customer Database Visualization 53
4.4 Order Database Visualization 54

List of Acronyms

API	Application Programming Interface
COM	Component Object Model
DB	Database
DBMS	Database Management System
RDBMS	Relational Database Management System
DDB	Deductive Database
POJO	Plain Old Java Object
POJI	Plain Old Java Interface
OO	Object Oriented
OOP	Object Oriented Programming
OOL	Object Oriented Language
ODBC	Open Database Connectivity
OLE DB	Object Linking and Embedding, Database
ISAM	Indexed Sequential Access Method
JDBC	Java Database Connectivity
OP	Orthogonal Persistence
ECRC	European Computer-Industry Research Centre
MCC	Microelectronics and Computer Technology Corporation

DJ	Deductive Java
VLDB	Very Large Databases
LB	LogicBlox
JNI	Java Native Interface
SQL	Structured Query Language
PL/SQL	Procedural Language / Structured Query Language
LINQ	Language INtegrated Query
IDE	Integrated Development Environment
XML	Extensible Markup Language

Chapter 1

Introduction

As soon as the computer was invented the potential was realized to use it for storing and retrieving information. In fact society has transitioned so well to using computers for the storage and retrieval of information that many people have dubbed the modern period in history "The Information Age" or "Digital Age". The title comes from the fact the the world's economy has shifted from physical goods to the possession and management of information. Goods and services are being offered electronically wherever possible, and information is being produced in, distributed via, and converted to digital mediums as fast as possible. This paradigm shift has resulted in the need for powerful information storage and retrieval systems commonly known as databases and database systems.

Database systems have become a ubiquitous part of everyday life. Though most people do not realize it, they interact with databases many times each and everyday. Whether visiting the bank, doing a web search, or even purchasing items at the store. Every bank transaction needs to be recorded, vast amounts of information on the web must be condensed and stored in such a way it can be quickly searched, and the grocery store needs to keep track of how many of each item is left in inventory. In light of these needs, it quickly becomes apparent that it is desirable for a computer to automatically perform these tasks, rather than to waste people's valuable time.

Due to the increasing volumes of global data and applications for database technology, the field is currently on the edge of a new era. No one familiar with database technology can argue that for the past 25 years traditional large vendor relational databases such as Oracle, DB2, and SQL Server have been the gold standard. Almost every large scale database development effort has used a traditional database on the back end. There are many reasons for the success of traditional relational databases, including incredible performance across a wide spectrum of applications, high reliability, familiarity of developers with the technology, and the support of large vendors.

1.1 Motivation For Change

Despite the overwhelming success and standardized use of traditional relational databases, some people are beginning to question whether they will continue to rule the future of database technology. For example, Michael Stonebraker of Massachusetts Institute of Technology (MIT) argues that the current mainstream database technology is no longer adequate to solely support the needs of data management systems. He points out that modern day relational technology can directly trace its roots back to the first relational database management system (RDBMS) of the 1970's [96]. He argues that even though the entire landscape of computing (users, processing power, applications, etc.) has changed since the 1970's, database vendors have chosen to stick with their traditional technology, or a "one size fits all" strategy [95]. He acknowledges that many improvements to the technology have been made, but states that this is for the express purpose of continuing to sell the technology without re-architecting it [96]. In his paper "One Size Fits All? Part 2: Benchmarking Results" [94], Stonebraker presents benchmarking evidence that "the major RDBMSs can be beaten by specialized architectures by an order of magnitude or more in several application areas", including:

- Text

- Data Warehouses
- Stream Processing
- Scientific and intelligence databases

Stonebraker is not alone in his assessment of the current state of database technology either. In an industry keynote at Very Large Databases 2007, Michael Brodie said "The confluence of limitations of conventional DBMSs, the explosive growth of previously unimagined applications and data, and the genuine need for problem solving will result in a new world of data management" [17]. He makes the claim that the growth of information between 2006 and 2010 will result in less than 5% of the world's data falling into the "relational" category. Brodie then went on to say that the data management world should "embrace these opportunities and provide leadership" for new data management solutions as the field of computing continues to evolve.

This assessment should not be a surprise to anyone familiar with the current computing landscape. As the use of computer and database technology has surged, so have the volumes, applications of, and need for data. It is therefore not surprising that the traditional technologies of the 1980's designed for general purpose use can be outperformed by solutions designed for special case data management scenarios. As Carrie Ballinger of TERADATA aptly points out, "dermatologists don't perform brain surgery, sprinters dont make good long distance runners, and gas-efficient commuter cars dont win stock car races. Nowhere does excellence in one category of endeavor translate to excellence in everything" [10]. The field of data management has seen many non-traditional technologies since its inception, however very few if any of these have been able to gain widespread use in industry. This thesis presents an alternative type of database technology called a "deductive database". A brief overview of deductive database technology and its applications is given, and then the research done to develop a bridge for providing deductive database functionality to the Java programming language in an intuitive way that hides the intricacies of deductive technology from the programmer.

1.2 Literature Review of Deductive Databases

A deductive database can be defined theoretically as logic programs that generalize the concept of a relational database [29], and from a more practical standpoint; a deductive database is an extension of a relational database that provides support for both recursive queries and views [21], and stored rules that enable inferences to be made based on the stored data [87]. Therefore it is imperative to establish solid definitions for both a "database" and a "database management system". Thus:

- A database represents some aspect of the real world, sometimes called the Universe of Discourse (UoD)
- A database is a logically coherent collection of data with some inherent meaning. Thus randomly assorted data does not comprise a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.[34]

and a database management system (DBMS) is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications [34].

The relationship between logic programming and deductive databases results from the fact that databases have been shown to be function free logic programs [42]. The increased power of deductive languages, in comparison to conventional database query languages such as SQL, can be taken advantage of in a variety of application domains, including decision support, financial analysis, scientific modeling, various applications of transitive closure (e.g., bill-of-materials, path problems), language analysis, and parsing. Deductive database systems are best suited for applications in which a large amount of data must be accessed and complex queries must be supported [82].

Deductive database systems generally divide their information into two categories:

- *Data* or *facts*, that are normally represented by a predicate with constant arguments. For example, the fact *parent(joe, sue)*, means that Sue is a parent of Joe. Here, *parent* is the name of a predicate, and this predicate is represented by storing in the database a relation of all the true tuples for this predicate. Thus, *(joe, sue)* would be one of the tuples in the stored relation [86].

- *Rules* or *program*, normally written in the form

$$P \leftarrow Q_1,...,Q_n$$

  This rule is read declaratively as "Q_1 and ... and Q_n implies P.

Databases of this form are termed Datalog databases [105]. The data or facts are referred to as the extensional database (EDB), and the rules are referred to as the intensional database (IDB) [86].

It is important to observe that despite the increased power that deductive databases offer, they never made their way into the realm of commercial technology. There may be many contributing factors to this, including the fact research on deductive databases occurring in parallel with relational databases resulted in a large number of contributions being applied to relational database implementations. Through the years, relational databases have continued to add support for more and more scenarios and types of data through features such as triggers, stored procedures, and views to name a few. In light of the failure of deductive database to make inroads into industry, and the fact that deductive databases can be defined as an extension of a relational database, it is important to understand a basic history of database evolution (both relational and deductive).

Table 1.1: Major RDBMS Releases

Year	RDBMS
1976	System R (IBM)
1978	Oracle
1983	DB2 (IBM)
1987	SQL Server (UNIX Version)
1988	SQL Server (OS/2 Version)
1993	Microsoft SQL Server
1998	MySQL (Windows Version)

1.2.1 Database Evolution

Early database systems began to appear in the mid-1960's and fell into three paradigms: hierarchical systems [101], network model based systems [18], and inverted file systems. Two major short comings of these systems were they that failed to separate the conceptual relationships of data from its physical storage, and only provided procedural programming language interfaces [34]. The first relational database system was published by E.F. Codd of IBM Research, San Jose, California in 1970 [19]. In this model, the physical storage of data was separated from its conceptual representation, and a high level query (logic) language in the form of relational calculus and relational algebra was introduced, that allowed the programmer to specify what "was to be done", and the database system to decide "how it was to be done". This query language allowed programmers who were not computer specialists to write declarative queries, and the computer to answer those queries. The subsequent development of syntactic optimization techniques [104, 106] permitted relational database systems to process queries with efficiency competitive to existing hierarchical and network implementations. Table 1.1 gives a historical overview of major RDBMS releases.

Even though relational databases used a logic language in relational calculus, relational calculus was not formalized in terms of logic [75]. It was not until 1984 that Raymond Reiter first formalized databases in terms of logic by observing that relational databases make many underlying assumptions about their data [89]. Specifically, he built off his previous work observing that with respect to negation, an assumption was being made that facts

not known to be true are assumed to be false. This assumption is called the *closed world assumption* (CWA) [88]. A second assumption of relational databases is the *unique name assumption*, which states that any item in a database has a unique name, and individuals with a different name are not the same. The last assumption made, is the *domain closure assumption*, which states that there are no other individuals than those in the database.

Based on these assumptions, Reiter stated that a relational database is a set of ground assertions over a language Ł together with a set of axioms–the language Ł does not contain function symbols. The formalization of relational databases in terms of logic then permitted the definition of a query and an answer to a query to be defined precisely. A query is a statement in the first-order logic language Ł. *Q(a)* is an answer to a query *Q(X)*; over a database *DB* if *Q(a)* is a logical consequence of *DB* [75].

1.2.2 Brief History of Deductive Databases

The idea for using logic with databases was born from work by JL Kuhns [61, 62, 63] in the 1960's including a computer question and answer system [61] published in 1967. Other notable contributers of the 1960's to the use of logic and deduction in databases were Levien and Maron [65, 66]. Since by definition deductive databases are logic programs operating on a relational database, the development of logic programming cannot be separated from the history of deductive databases. Hence another major influence was the work of Kowalski [60] forwarding logic programming as a language, and the development of the first Prolog interpreter by Colmerauer [20].

The idea of deductive databases as a field was born in 1976 when Jack Minker, Herve Gallaire, and Jean-Maire Nicolas had the idea for a workshop called "Logic and Databases"; held in Toulouse France [75]. The product of the workshop was a book edited by Gallaire and Minker titled "Logic and Databases" [37]. Following this were two more textbooks [43, 44], which were the products of subsequent workshops on logic and databases.

Another key work which served to forward the field was a publication by Minker, Gallaire, and Nicolas in 1984 surveying current work in the field at that time titled "Logic and Databases: A Deductive Approach" [74], followed by a second paper responding to critics of the first paper titled "Logic and Databases: A Response" [38]. It was at this time that three major deductive database projects emerged. Work at the European Computer-Industry Research Centre (ECRC) under the direction of Jean-Maire Nicolas began in 1984. The Logical Data Language (LDL) project was begun at the end of 1984 through the newly created Microelectronics and Computer Technology Corporation (MCC) research consortium, and in 1985 Stanford began the NAIL! (Not Another Implementation of Logic!) project.

Work by the ECRC can be divided into two major phases, an initial phase from 1984 to 1987, and a secondary phase from 1988 to 1990. The initial phase of research ('84-'87) led to the development of early prototypes of the deductive systems QSQ, SLD-AL, QoSaQ, and DedGin [108, 109]), integrity checking for SQL in Soundcheck by H. Decker [31], and a combination of deductive and object oriented database ideas in KB2 by M. Wallace [112]. The second phase ('88-'90) led to the more functional prototypes: Megalog by J. Bocca [15], and DedGin [111] and EKS-V1 [110] by Vieille. The EKS system supported integrity constraints and some some forms of aggregation through recursion.

The LDL project was perhaps the most successful deductive database project in history. It began at the end of 1984 in the the Database Program of the MCC. The objective was to build a database system with a powerful language for developing complex data-intensive and knowledge-based applications [116]. The name LDL first appeared in a paper by Tsur and Zaniolo [102] in VLDB 1986; the paper describes the main constructs in the language, such as complex terms and recursion, sets, set aggregates, and database schema declarations. The first prototype completed in 1987, compiled LDL programs into an extended relational algebra language designed for a non-existent MCC parallel database machine. Therefore, in 1987, the deductive computing laboratory of MCC began development of a new stand-alone deductive database system for LDL which was completed in 1989. The completion of the LDL prototype allowed the development of new applications that revealed the potential of

LDL in several areas including: the rapid prototyping of information systems, intelligent middleware and information brokering, data cleaning and conversion, and data mining. The subsequent delivery of LDL technology to MCC's shareholders resulted in the development of LDL++ in 1992 [118].

Beginning in 1992, the LDL++ system was deployed in several middleware applications; including as an important component of the Infosleuth [56] technology suite that was used to construct intelligent agents that supported the semantic interoperability of distributed databases and heterogeneous information resources. LDL++ continued to be developed at UCLA [115] resolving research issues limiting the language's power. Finally a new version of LDL++ which included support for non-monotonic recursion and generalized user-defined set aggregates was developed in 1999 by Haixun Wang [113].

The NAIL! project began at Stanford in 1985. It was based off of J.D. Ullman's paper "Implementation of Logical Query Languages for Databases" [103], with the goal of studying the optimization of logic using the database-oriented "all-solutions" model. The first paper on Magic Sets [11] and regular recursions [77, 78] resulted from this project in collaboration with the MCC group. Many of the important contributions for handling negation and aggregation in logical rules were also made by the project. Stratified negation [40], well-founded negation [41], and modularly stratified negation [90] were also developed in connection with this project. An initial prototype system named YAWN! "Yet Another Window on NAIL!" [76] was built, but eventually abandoned because the purely declarative paradigm was found to be unworkable for many applications. The revised system used a core language, called Glue, which was single logical rules, wrapped in traditional language constructs such as loops, procedures, and modules [80, 32, 33].

Many other deductive database implementations have arisen over the years, of which an excellent overview can be found in the book "Applications of Logic Databases" [82] edited by R. Ramakrishnan. These include the Aditi project [107] which began at the University of Melbourne in 1988, the CORAL project [84, 85] begun at the University of Wisconsin

in 1988, and the DECLARE project [59] at MAD Intelligent Systems which ran from 1986 to 1990, and was perhaps the first attempt at a commercial deductive database. Beginning in 1994 several papers were written by prominent researchers in the field reflecting on the history of deductive databases and outlining their views for future directions [86, 75, 87]. These papers in many ways were indicative of the deductive database research climate at the time. The field had been in existence for almost 20 years, and no major commercial implementations had yet to materialize. Those prominent in the field perhaps began to see the beginning of the end, and therefore naturally began reflect on the technology's history as well as to make one last push for continuing research in the field. It was however at this time that they began to fade in popularity, and publications in the area began to cease as researchers moved into new areas. The field did continue to see some minor activity for the next few years with publications in 1998 [117, 21], and a new version of LDL++ released in 1999 at UCLA [113]. These were however based on continuations and conclusions from previous work, and no significant research has been done in the field since 2000.

1.3 Statement of the Problem

Modern society is becoming increasingly dependent on data and data management. As volumes of data balloon, and the dependency of companies on this data for success continues to grow, an opportunity has arisen for companies to move into the data management market with unique solutions. However as these companies develop new technologies that model data in domain specific ways they introduce problems for their potential customers. Large vendor relational databases have been the industry standard for the past 25 years, and as such have many well known constructs and conventions. Unfortunately when modeling data in a new and unique way, existing standards may be inadequate. For example, if data is not stored in a standard relational database, writing queries in the Structured Query Language (SQL) may be highly impractical or even impossible. Unfortunately companies prefer to stick with known technologies that have well documented and understood development costs.

In order for a new player in the market to succeed they must be able to convince clients and developers that applications are easy to build and maintain with their platform. Thus it becomes imperative to provide constructs in industry standard languages that enable developers to quickly learn and maintain new technologies by hiding many of the underlying intricacies in a middleware system.

1.4 Statement of Objectives

The objective of the research herein is the development of a bridge between a deductive database and Java programmers. The bridge shall consist of a library of annotations available to Java programmers for mapping Java interfaces to data stored in the deductive database, and a middleware system that handles all communication between Java applications and the database. The system should provide tools for allowing programmers to both generate annotated Java interfaces from an existing database, and to create and deploy a new database (including calculation rules for individual fields) from Java code that is properly annotated using the provided annotations library. The system should allow a programmer with limited knowledge of a deductive database to quickly develop and easily maintain a software system written in Java that uses a deductive database for its data management system.

1.5 Overview of Thesis

Creating a bridge that consists of client annotation libraries and a middleware system provides benefits to both the customer and deductive database vendor. Chapter 1 has introduced database technology in a general way, both its current status and the need for future development. Chapter 2 gives an overview of LogicBlox technology, the problem which this thesis solves, and the design objectives and challenges of implementing a solution. Chapter 3 reviews related work, Chapter 4 details the solution to the problem described in Chap-

ter 2, and Chapter 5 summarizes the contributions of this thesis and outlines future work directions.

Chapter 2

LogicBlox: A Real World Need

LogicBlox is a company that has embraced the challenge of producing new database technologies to meet the global demand in data management. They are based out of Midtown Atlanta, and have developed a deductive database technology that they use as a platform for planning applications. The company was the result of a group of database developers in conjunction with an entrepreneur who has founded several successful companies. The database developers saw first hand that current database technologies are not adequately meeting the needs of the ballooning data management industry, and decided an opportunity existed to release a technology that met these needs.

2.1 LogicBlox Implementation Overview

Some of the key features of LogicBlox's implementation include:

1. Datalog not SQL is used for the database language
2. The database engine is implemented entirely in C++

3. Java libraries exist for the database, and communicate with the underlying C++ through JNI interfaces

4. Data is stored in a column oriented manner

The domain of planning applications that LogicBlox targets introduces many unique uses for data and therefore has very complex challenges associated with data management. It is the above features that combine in a unique way to allow LogicBlox to demonstrate much better performance in this area than traditional RDBMS's such as Oracle. However, it is the very features that make them outperform traditional technologies that pose problems for enterprise developers.

Enterprise developers are incredibly familiar with traditional large vendor relational technologies. They offer the benefits of a well known, well understood data storage paradigm. Development time, costs, and implementation trade-offs are well understood and documented. Tools for data reporting and analysis are widely available, and standardized interfaces (ODBC, OLE DB, JDBC) for database access are available. When attempting to market a new database technology such as LogicBlox these features do not exist for the current product, and are therefore challenges that all must be addressed.

Enterprise developers need to be convinced that the benefits of moving to the new technology offset the familiarity, standardization, and all of the other benefits that come with traditional technologies. One of the ways to better convince enterprise development teams to switch to newer technologies better suited for their domain areas is to offer tools in the familiar object oriented programming paradigm that improve programmer productivity by reducing the learning curve for the technology and reduce software maintenance costs in the future.

In order to provide such tools, it is imperative to understand why the LogicBlox technology works well the domain of planning applications, and what challenges this presents for a programmer familiar with traditional RDBMS's. One of the key things about planning applications is that they tend to use data in a "read-mostly" environment. It is the column

oriented data storage feature of the LogicBlox engine that allows for great performance in the "read-mostly" environment as opposed to traditional "row-oriented" data storage of traditional RDBMS's. Understanding this data storage model is a key challenge for developers unfamiliar with the technology to overcome, and introduces several unique challenges when providing tools for interfacing object oriented languages with the database. Therefore an overview of column oriented data storage is presented below.

2.2 Overview of Column-Oriented Databases

System R, the first relational database implementation to use SQL, was released by IBM Research in 1976 [3], and all current popular relational DBMS's can trace their roots back to System R [16]. For example, IBM's DB2 (1983) directly descended from System R, Microsoft's SQL Server was purchased in 1993 from Sybase, who released their System R based implementation in 1987 [51], and Oracle's first release (1978) borrowed System R's user interface [28].

It is important to point out that all three systems mentioned above (DB2, Oracle, SQL Server) were designed over 25 years ago, when hardware constraints and DBMS markets were much different than they are today. Processors are thousands of times faster, main memory is thousands of times larger, disk capacity has increased exponentially, and when relational DBMS's were created there was a single DBMS market–business data processing. Since then the number of new markets has exploded, including data warehouses, text management, and stream processing, all having very different requirements than business data processing [96].

In summary, the current RDBMSs were designed for a single data market and architected for different hardware characteristics. Thus, they include the following System R architectural features [96]:

1. Disk oriented storage and indexing structures

2. Multi-threading to hide latency
3. Locking-based concurrency control mechanisms
4. Log-based recovery

It is important to note that these systems have been extended and improved, including support for compression, shared-disk architectures, bitmap indexes, support for user-defined data types and operators, etc. However, while these systems have been extended and improved to accommodate new data needs, they have not been re-designed to optimally address them.

Current relational technologies fail to scale well to new data markets with large volumes of data due to their record-oriented storage implementations. In record-oriented implementations the attributes (the columns in a row) of a record are placed together in storage. With this row store architecture, a single disk write pushes all of the fields in a single record to the disk. Hence, high performance writes are achieved, and therefore DBMS's with a row store architecture are write-optimized systems. When developing Sybase IQ, the development team realized that historically, databases have been designed around the requirement to support large numbers of small concurrent updates. Consequently they noted that when the data writer is given preference, the internal design generally adheres to the following rules [68].

1. Data should be stored in rows to enable a single disk i/o to write the modified data out to the table.
2. Data should be stored on small page sizes to minimize the i/o cost and portion of disk locked for exclusive access.
3. The database must support the imperfect notion of isolation levels to enable reports to run in acceptable time while negotiating held locks.

4. Few columns should be indexed because locks on tree structures may deny access to more rows than row-page locks and increase the time locks are held.

5. Data pages should typically not be compressed because of poor amortization of the compression cost. Rows typically do not compress well because of the mix of data types stored adjacently.

6. Adding or dropping a column or index may be expensive since page storage may need to be updated for all rows.

7. Searched update statements may be relatively expensive since the entire row must be read and written for a single column to be updated.

However, in modern analytical systems, update performance is often less important than the ability to quickly process queries over large volumes of data. Using traditional RDBMS's in these cases can result in the performance of even simple user questions, such as "How many of my female customers in Mass. bought product A?", taking hours to run [36]. According to Stonebraker of MIT, "based on this evidence, one is led to the following conclusions: RDBMSs were designed for the business data processing market, which is their sweet spot, and they can be beaten handily in most any other market of significant enough size" [96].

Therefore the Sybase IQ designers started with the question "What would a database look like internally if it were designed from the ground up for complex analytics on massive amounts of data"? They noted that typical analytical queries access relatively few columns of the fact tables but access a large percentage of the rows stored in tables [68]. Hence, systems oriented toward ad-hoc querying of large amounts of data should be read-optimized. In such environments, a column store architecture, in which the values for each single column (or attribute) are stored contiguously, should be more efficient [93]. This efficiency has been demonstrated in industry by products like Sybase IQ [68], Kdb+ [98], SenSage [91], and LogicBlox [67].

With a column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid reading irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a significant performance advantage [93]. However, there are several other major distinctions that can be drawn between an architecture that is read-optimized and one that is write-optimized. Current relational DBMSs (write-optimized) were designed to pad attributes to byte or word boundaries and to store values in their native data format. It was thought that it was too expensive to shift data values onto byte or word boundaries in main memory for processing [93]. However, while CPU performance and available cache memory has increased dramatically, disk performance has not kept up and, consequently, disk-bound performance is typical for many analytics [99]. So it makes sense to trade CPU cycles for disk bandwidth, which appears to be very profitable in a mostly read-only environment. Hence were database storage designed for complex analytics, many of the rules for the write-based design are reversed in the following ways [68]:

1. Data would be stored in columns, not rows, so only the columns required to answer a query need be read in effect every possible ad-hoc query could have performance akin to a covering index. Since data is only written once but read many times any increase in load time would be more than offset by improved query performance. Column storage would enhance main cache efficiency since commonly used columns would be more likely to be cached.

2. Data would be stored in a large page size so that many cells of a column can be retrieved in a single read and the larger page size plays to technological changes in how physical disk reads are structured. Traditional DBMS cannot use large pages sizes since each read drags along unneeded columns in the row.

3. Every column could be indexed: data will be written once and read many times so the cost will be amortized. Column storage at the physical layer greatly simplifies parallel

index updates and adding an index requires only that column to be read, not the entire row.

4. Data could be compressed on disk. The compression cost would be well amortized and the homogeneous data type of a stored column would offer optimal compression.

5. Adding or dropping a column or index to reflect changing business requirements would be cheap, as no other data would be accessed.

6. Searched updates would be relatively cheap since only the columns being modified would need to be read and written.

2.3 Objectives

Armed with an understanding of column oriented data storage and why it performs well in the realm of planning applications for LogicBlox there is no doubt that the technology has potential. Kx Systems (makers of Kdb+) states that the firms that are first to market with new technologies are the ones who can maintain and expand their competitive strategies [99]. Therefore it is the aim of this work to provide tools to Java programmers that allow for the easy adoption, development, and maintenance of applications that use the LogicBlox database engine on the backend.

The primary object of this project as illustrated in Figure 2.1 is to provide Java programmers with deductive database functionality through a system named "DJ". The system should meet several goals and requirements, including:

- The system shall use only standard Java language constructs
- The system shall provide a library of Java Annotations available for mapping Java interfaces to data stored in the deductive database

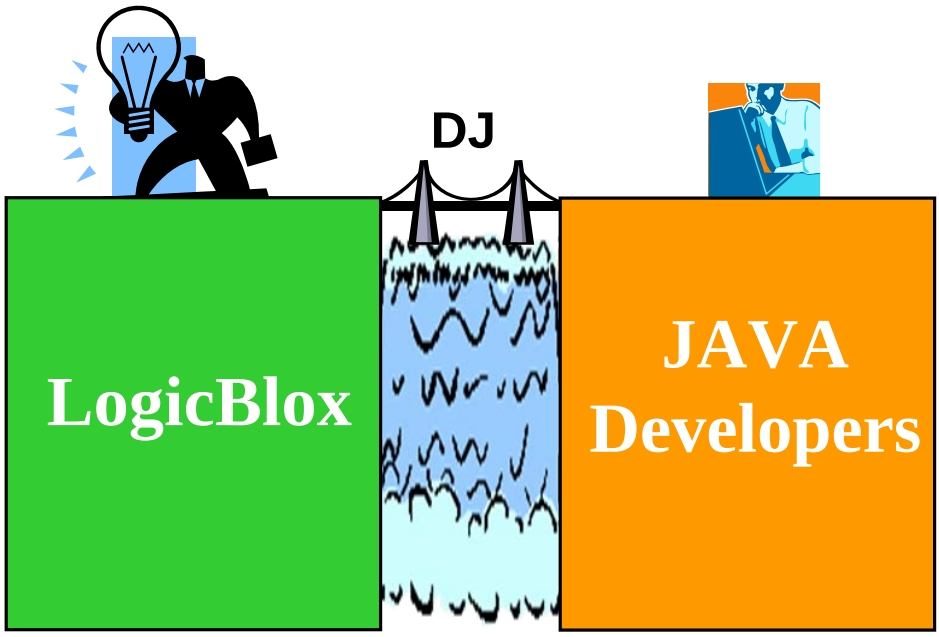

Figure 2.1: Project Visualization

- The system shall include a middleware layer that handles all communication between Java applications and the database.
- Tools shall be provided that allow programmers to generate annotated Java interfaces from an existing database.
- Tools shall be provided that create and deploy a new database (including calculation rules for individual fields) from properly annotated Java code
- The system will allow a programmer with limited knowledge of a deductive database to quickly develop a software system written in Java that uses a deductive database.
- The system shall reduce the maintenance costs of a software system that interacts with a deductive database.

Java was chosen as the target programming language due to the ubiquity of it's use in industry. In 2006, Sun Microsystems estimated that there were 5 million Java developers

worldwide [72], and that number has certainly continued to grow. Therefore when developing a bridge technology for an enterprise language, it made sense to target Java first.

As is common with new technologies, the LogicBlox API libraries have already been through several major update phases since their inception, and code dependent on those libraries sometimes will not work without modifications. The middleware layer that separates the developer's Java code from the database interaction should reduce the maintenance effort by eliminating code's sensitivity to API updates by the database vendor (LogicBlox in this case).

The idea for this Java-LogicBlox bridge is that it will learn from and leverage existing technologies in order to determine what the best design principles and approaches are. The system as a whole must be easy to learn, and must not impose an unreasonable burden on the programmers to either learn, program with, or maintain.

Thus it is also important to note when developing a system such as this, what the goals are not. The goals of this system are not to implement JDBC, ODBC, OLE DB, or any other connectivity standard for the LogicBlox engine. JDBC, ODBC, and OLE DB are all standardized database connectivity interfaces that were developed through the cooperation of large corporations, and provide standardized API access to databases rather than vendor specific interfaces. They do solve many of the problems that the LogicBlox middlware bridge will solve, but it is not the goal of the LogicBlox middleware bridge to adhere to the standards of or provide a working implementation of existing connectivity technologies. Nor is the goal to provide a full implementation of Orthogonal Persistence (such as Java Hibernate) for LogicBlox. Much like the connectivity standards, Java Hibernate and other Orthogonal Persistence systems have had many years of development and industry input into them to offer the full range of features and optimized performance they currently provide. This system is designed to be a working prototype to demonstrate the feasibility and principles needed for development, but it is not meant to be industry standard, or enterprise quality.

2.4 Challenges

Many challenges exist when attempting to bridge two distinct technologies. As has been previously stated in the "Objectives" section, the purpose of this project is to provide tools to Java programmers that allow for the easy adoption, development, and maintenance of applications that use the LogicBlox database engine on the back end. Challenges stem from both the developer perspective and the technological perspective.

From the typical Java developer's perspective database interaction happens in SQL. Unless other tools such as Hibernate are used to automatically generate the SQL for the programmer, the programmer is used to generating SQL queries as strings, creating a JDBC object and passing string value to the JDBC driver. It does not matter what the underlying engine is, the results will be the same. JDBC drivers should hide any differences in the underlying engine's handling of SQL, and accept standardized query strings. Updates and inserts will return an integer value to indicate success or failure or throw an exception. Queries are much the same, they return a standard result set on successful execution or throw an exception in the event of an illegal statement. Once a result set is returned, the programmer has full discretion to do whatever they want with it.

LogicBlox does not store data in a traditional "table" format, but rather data is stored inside "predicates" which are tuples that consist of a key space that is the set of foreign keys from other predicates required to uniquely identify that tuple, and a single value attribute. This follows the column-oriented data model that was presented above, and offers many performance advantages. However this is a foreign concept to most developers who consider "table" to be synonymous with "database". Currently tools exist for mapping database tables into object oriented languages (e.g. Java Hibernate), but they all do so with the "row-oriented" model in mind. DJ needs to be able to leverage the advantages of the column-oriented data model while intuitively mapping the model into the well understood object oriented paradigm. In doing so, it should not completely hide the data store structure to make developers feel like they are operating in the row-oriented world, but should prevent

developers from struggling to understand the column oriented storage model.

Due to the predicate data modeling, in the LogicBlox world queries are not written in SQL, but Datalog–a declarative logic programming language that is a subset of Prolog. Declarative logic programming lends itself very well to this data storage model, but logic programming is an unfamiliar paradigm for the average developer. So in order to write query intensive applications for LogicBlox, they must not only learn the LB Java API and database structure, but they must also learn Datalog itself. So the next challenge presented for the DJ system is how to hide enough Datalog from programmers who are unfamiliar with the technology so they can begin to develop applications and query the database, but not remove the benefits of logic programming from them. DJ would like to address the issue of storing and passing queries inside Java *String* objects where possible. Strings are incredibly difficult to debug, because it is the underlying engine that must handle all of the parsing and compilation, therefore errors can be incredibly hard to pinpoint and debugging can become a nightmare. So DJ must figure out how not only to hide as much Datalog from the programmer as possible while not withholding the benefits of Datalog, but how to provide the benefits of Datalog without the programmer needing to write queries as strings at all.

A key benefit of the LogicBlox implementation resulting from the combination of Datalog and predicate data modeling is the ability to easily apply Datalog programs calculating value fields for predicates. This ability is currently achieved in existing database systems using rule engines or stored procedures. The ability to apply basic logic programs to calculate other predicate value fields is very powerful and has the potential to greatly reduce the amount of code needed to accomplish basic tasks. Especially when considering that in current RDBMS's the queries are written in SQL, and the calculation code must be written in something else (such as PL/SQL, or a rule engine). The problem that is faced by the DJ system is how can a programmer not familiar with logic programming be exposed to deductive functionality in the form of Datalog programs on predicates while learning a completely new technology. Then how can this be modeled in an object oriented language in such a way that it is not just strings passed into a driver's interface as parameters?

The final challenge that DJ attempts to solve is how to create an interface with a non-engine specific flavor. The benefits of this are many. For an emerging technology such as this, a non-engine specific flavor makes developers feel that it is just normal Java development. If Java developers feel that the development has a very standard natural feel, in a non-engine specific manner, it opens the potential for small companies to push their database interface as a selling point for their technology. Also the further removed the interface is from database specifics, the less likely it is that any underlying database API changes will affect user code. The user will only need to update their middleware system that has been provided by the vendor (LogicBlox in DJ's case).

Chapter 3

Related Work

Before the challenges that developing the DJ system poses can be solved, it is important to evaluate related work that has been done in the realm of interfacing databases with object oriented languages. DJ must learn from both historical solutions and current "state of the art". The study of both historical solutions and current state of the art is necessary to understand if existing technologies (whether "as-is" or with reasonable modifications) will provide an adequate solution to the challenges faced by interfacing Java with the LogicBlox database engine. Historical techniques must be studied in order to understand the issues that were addressed when developing similar solutions as DBMS's evolved and moved widely into the realm of computing. Studying historical solutions also allows DJ to leverage the lessons learned by previous development efforts to solve a similar problem. Current state of the art must be studied to understand what is currently available for the development of DJ, how DJ is related to and different from current state of the art, as well as to glean ideas and leverage lessons learned from research with current technologies.

The rest of this chapter overviews existing work in the area including database connectivity, orthogonal persistence, Microsoft's LINQ project, and then gives an evaluation of the applicability of this related work to the DJ system.

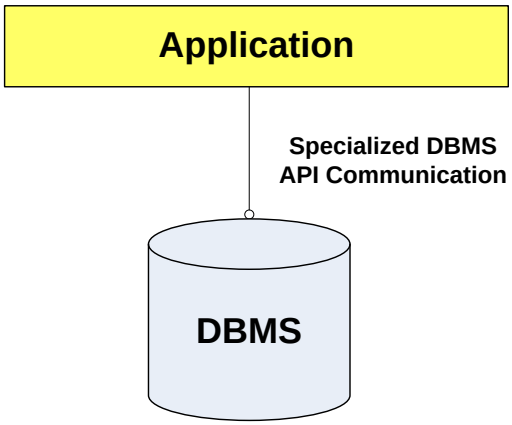

Figure 3.1: Database communication through a vendor API

3.1 Database Connectivity

Corporations and application developers from every realm of computing have always needed to interface domain specialized applications with database management systems. Good software engineering principles dictate that with the advent of advanced DBMS systems, software written for other purposes should leverage the advanced functionality of a well tested and engineered DBMS, rather than re-inventing (and most likely poorly at that) an in-house DBMS. The design principle is simple, DBMS vendors develop and maintain these systems for the express purpose of data management. Therefore enterprise developers that specialize in other fields will never be able to develop or maintain a DBMS that is the quality of a DBMS vendor's product. Plus the cost of purchasing a DBMS implementation from a vendor is ultimately cheaper than the cost of paying in-house experts to develop and maintain another complex code base that does not directly contribute to the enterprise's reason for existence.

The use of a third party vendor's DBMS, however introduces the challenge of interfacing applications written by a company's enterprise application developers with the third party DBMS. The traditional solution for this is relatively simple, the third party vendor provides an Application Programming Interface (API), which is a library of calls, in a predefined

language so other programs "know how" to talk to the vendor's software. This method of communication is depicted in Figure 3.1, and on the surface may appear to be a very logical solution that raises no immediate concerns. However a closer look at the longterm software engineering implications of this paradigm will quickly raise a host of considerations, including:

- What if a company needs to change their underlying DBMS?
- What if a DBMS vendor releases a newer version that requires API changes?
- What if a single application needs to consume data from multiple DBMSs?
- What if the production environment DBMS is not available to application developers?
- Once a company selects a DBMS, are they forced to stick with that DBMS for the life of the applications consuming that data?
- What if a better DBMS is released into the market? Will the cost of converting existing software to use the new DBMS prevent a company from migrating to the better technology?
- How hard will it be for the in-house development team to learn a new technology if the company chooses to make a change? Will this require hiring new employees who are experts for the new system, while retaining current experts to maintain existing systems?

With these considerations in mind, it quickly becomes apparent that it is desirable for a communication layer called "Database Connectivity" to be put in place between enterprise applications and DBMS's as depicted in Figure 3.2. By providing a layer of abstraction between the application and database, it becomes possible to develop a standardized communication protocol between applications and the database connectivity layer, and then the database connectivity layer can handle the specialized database API calls. Hence if a change

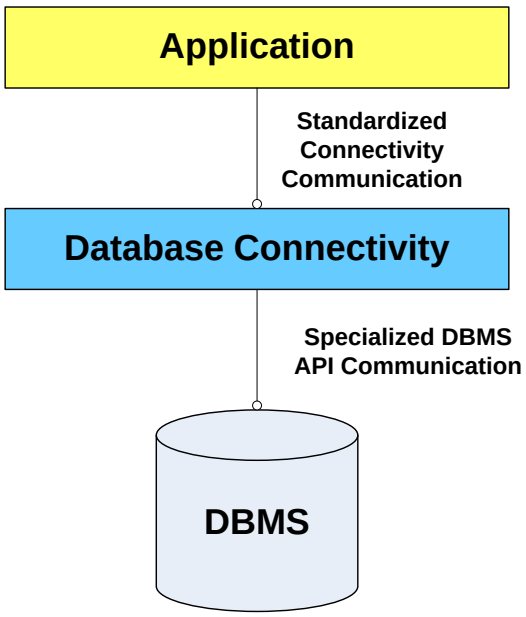

Figure 3.2: Database communication through a vendor-independent standard

in the underlying DBMS is required, only the connectivity layer will need to be replaced, and the enterprise application will not need to be refactored.

Whenever the idea of adding a layer of abstraction is put forth in software, the most important questions to ask are what are the benefits and drawbacks, and who are the stakeholders? In the case of database connectivity these questions must be answered from the perspective of the three stake holders in the process: application users, application developers, and DBMS vendors. The following three subsections give an overview of the motivation and need for database connectivity from each unique perspective. Many of the design considerations and challenges can be traced back to Microsoft's development in partnership with others of the first database connectivity standard "Open Database Connectivity" (ODBC) as presented in "Inside ODBC" [39].

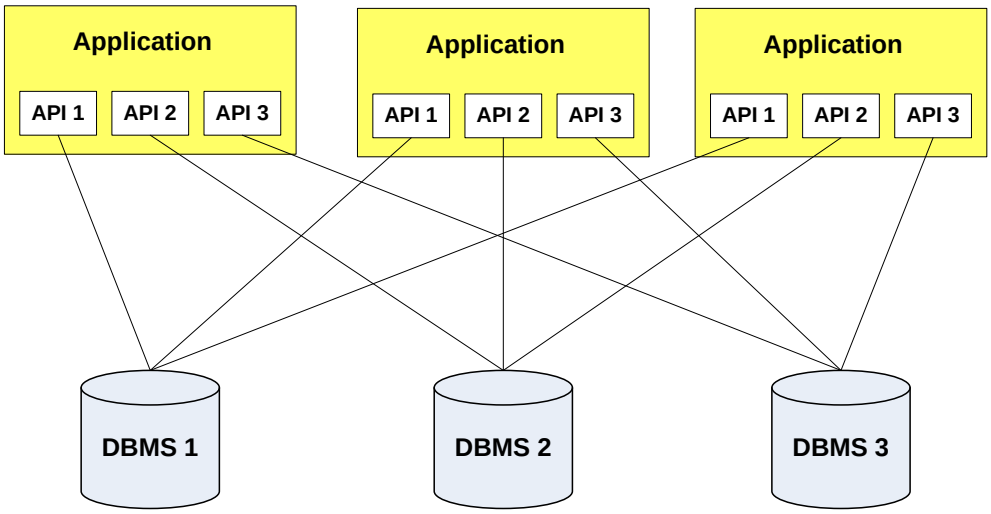

Figure 3.3: The user's perspective of the need for database connectivity

3.1.1 Users

Users are defined as people that use computers to make their jobs or businesses more productive. Most of the time users are domain experts for their particular field, but only have a basic knowledge of computers and computing systems. They care about what can be accomplished with the computer, and are not necessarily concerned with the details about how the computer accomplishes it. So long as their tasks are accomplished quickly and efficiently. Figure 3.3 shows the need for a database connectivity layer from the user's perspective. The following is a list of requirements for database connectivity from the users perspective:

- **Flexible extraction tools:** Users need to pull data from large databases into tools that they are familiar and comfortable with. These tools are then used by the domain expert to analyze and use the data for business processes, and should allow the extraction of data from any underlying databases.

- **Applications can be developed and modified quickly:** When a new DBMS is brought online to manage corporate data, business domain experts quickly need tools to access these data stores. If an existing system is updated or changed, they need

their current tools to be able to quickly adapt so they can continue to do their jobs.

- **Works with existing technologies:** When a standard is formalized, it is not acceptable to require the replacement of current software and hardware.
- **Exploits DBMS specific features:** Individual DBMSs often have unique strengths and weaknesses, and are chosen specifically for those reasons. When a connectivity standard is introduced, it cannot simply default to a "lowest common denominator", meaning it must allow applications to leverage the benefits of each unique DBMS.
- **High Performance:** Users care about how long it takes to do something. A connectivity standard must offer performance that is competitive to communicating through vendor specific APIs.
- **Simplicity:** Users are domain experts, not computer experts. They need to be able to use software to meet their business needs without understanding the complexities of the underlying system(s). Whether a database is on the same computer, or connected via the network, the user should to see the same interface.

3.1.2 Application Developers

Application developers are the people who write the applications for users, or develop tools for application developers (development tool developers). These are generally in-house software experts for companies, and know the underlying technologies of a company very well. They often are familiar on a high level with the companies business needs, and at a very detailed level with how the companies information technologies are structured. They are concerned with quickly meeting the needs of users, and maintaining the company's tools and information management systems.

They are perhaps the biggest target and beneficiary of database connectivity standards, as they are responsible for writing the software that communicates with DBMSs, and are

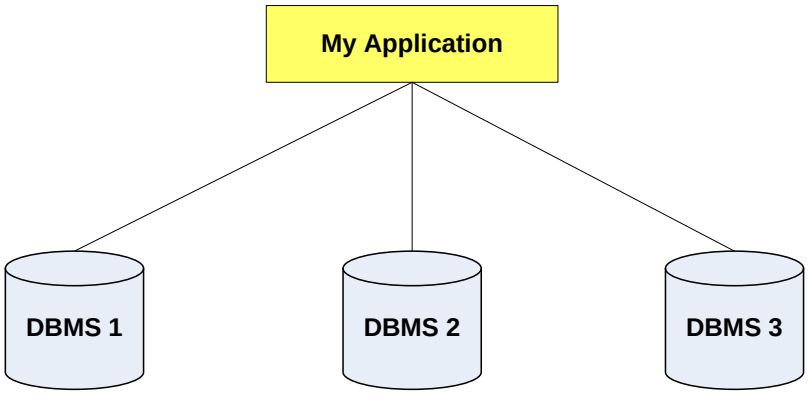

Figure 3.4: The application developer's perspective of the need for database connectivity

generally responsible for making decisions about which DBMS to use. As such, they are subject to the same requirements and considerations as the users, and also have a list of their own requirements. Figure 3.4 shows the need for a database connectivity layer from the developer's perspective. The following is a list of requirements for database connectivity from the developer's perspective:

- **Eliminate DBMS vendor lock-in:** Developers want to be able to develop software than can communicate with different DBMSs using the same API. They need the flexibility to upgrade underlying systems without the need for intensive updates to existing code bases in order to be compatible with new systems.

- **Allow for code re-use with different DBMSs:** Software systems for internal corporate use often have large similarities in the use and nature of data, and therefore introduce the possibility for re-using existing code modules. However the potential exists due to various underlying data requirements that the data sources for two different applications may differ, while the front end may be the same. Thus it is desirable to allow for code re-use with the ability to quickly reconfigure data source access.

- **Application prototyping that allows access to multiple DBMSs:** It is desirable for developers to be able to develop applications in a isolated environment. Meaning that initial phases of development should not depend on existing corporate resources.

However due to constraints such as cost and resource availability it my be impractical for a developer to have a deployment environment that mirrors the production environment. (Take the example where a production environment uses Oracle due of performance constraints, however because of the cost of an Oracle license, developers are required to prototype using SQL Server). Developers should be able to develop a working prototype with the technologies available to them and then quickly migrate code for the development environment to the production environment.

- **Hide complexities of data access in distributed systems:** Similar to the above requirement, it may be impractical for a developer to develop in a distributed environment. Meaning that even though the database server exists separately from the application host in a production environment, it may be much more efficient for a developer to create applications with a database server physically located on the application development machine. When it is time for this code to move to the production environment, the developer should quickly be able to reconfigure the application to access the database server through the network.

- **Full DBMS functionality and high performance:** Developers need to be able to take advantage of specific features of DBMSs. They may need to use several different DBMSs on the backend, and need their standard API to allow their applications to take advantage of the specific benefits of each DBMS, not be restricted to the lowest common feature set.

3.1.3 DBMS Vendors

Database vendors have a unique perspective when it comes to the concept of database connectivity. Their number one priority is to make sure that customers use their technology. Therefore on one hand it is appealing to force customers to commit to communicating through their specific API. The cost of re-architecting applications for a different vendor's

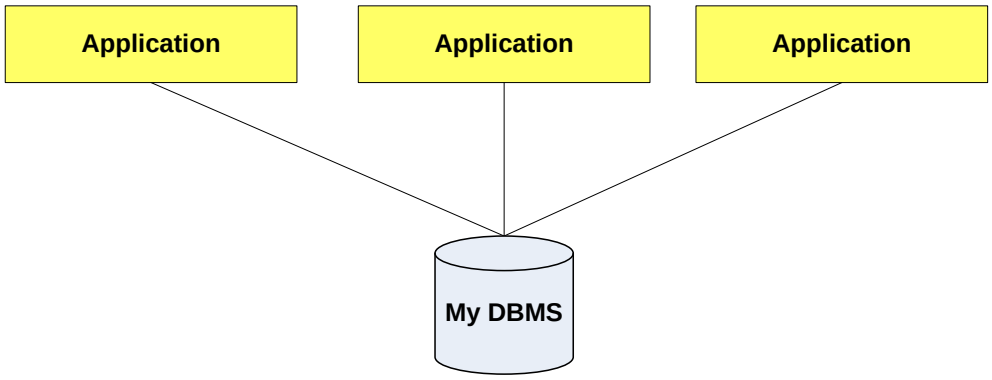

Figure 3.5: The DBMS vendor's perspective of the need for database connectivity

API means that once customers commit it becomes very hard for them to migrate to a different DBMS vendor; so customer retention is very easy without an industry standard for connectivity. There are however several compelling reasons for DBMS vendors to support the idea of a database connectivity standard. Figure 3.5 shows the need for a database connectivity layer from the DBMS vendor's perspective. The following is a list of requirements for database connectivity from the DBMS vendor's perspective:

- **Satisfy Customers:** Customers don't want to be locked into a specific vendor. Thus the real need for database connectivity comes from the requirement of keeping customers happy. Therefore if other vendors agree on a standard, and a particular one does not, clients will simply choose to go with vendors who comply with the standard. Also, with the client server model, DBMSs are very much separate from the applications that access them. Hence, the more people that write quality applications for a particular vendor's DBMS the more customers that are likely to purchase that DBMS. So it becomes incredibly important to a DBMS vendor to make developing tools for their product as easy as possible.

- **Comply with Standards:** Many customers (including the government, large companies, and international clients) require that companies comply with standards to bid on contracts. Thus if a standard exists, a company must comply with that standard

to gain more business.

- **Avoid "API war" marketing:** Instead of trying to market and advertise the benefits of a particular API, DBMS vendors can share the burden of advertising the technology, and then focus on marketing their implementation's particular benefits.

- **Allow for improvements:** By complying with a standard for database connectivity, vendors remove the need to support specific API calls. Thus if the implementation needs to evolve, the vendor does not have to consider how minor changes may affect the the ability of customer's legacy applications to communicate with their new implementation.

- **Compete for customers:** While migrating away from a vendor specific API offers the drawback of not forcing customers to commit to that vendor's technology, it offers the benefit of a vendor competing for new customers that have previously chosen an alternate technology. Because connectivity is standardized, the burden on a customer's development team is minor if a vendor can sell the idea that their implementation is a better fit for that customer.

3.1.4 Connectivity Implementations

The principal of database connectivity implementations is simple. Applications developers follow a standard set of API calls to interface with a connectivity driver, and the database vendor provides the connectivity driver for their specific database. The vendor's connectivity driver that implements the standardized API methods for incoming method calls, and then handles outgoing calls to the DBMS via the vendor's specific DBMS API. This way the user has "write once DBMS independent code" for database access.

The need for connectivity was answered in 1992 through Microsoft's Open Database Connectivity (ODBC) API. When considering the realm of database connectivity there are 3 major connectivity standard implementations that merit an overview: Microsoft's ODBC

and "Object Linking and Embedding, Database" (OLE DB), and Sun Microsystems' Java Database Connectivity (JDBC).

ODBC

The Open Database Connectivity API ODBC was released as a standard in 1992 resulting from the partnership between Microsoft and Simba Technologies [100]. The ODBC API provides the following: [92]

- A library of ODBC function calls that allow an application to connect to a DBMS, execute SQL statements, and retrieve resutls
- SQL syntax based on the X/Open and SQL Access Group (SQG) SQL CAE specification
- A standard set of error codes
- A standard way to connect and log onto a DBMS
- A standard representation of data types
- A standard way to convert data from one type to another

An ODBC application has four main components in its architecture[92]: the application, the ODBC driver manager, the ODBC driver, and the data source. Figure 3.6 shows the relationship of ODBC components. The ODBC Driver Manager loads and unloads the application-requested drivers, along with processing some of the ODBC function calls. The ODBC driver processes most of the ODBC function calls, submits SQL requests to a specific data source, and returns the results to the application. Also if necessary, the driver modifies the application's request to conform to the correct DBMS syntax.

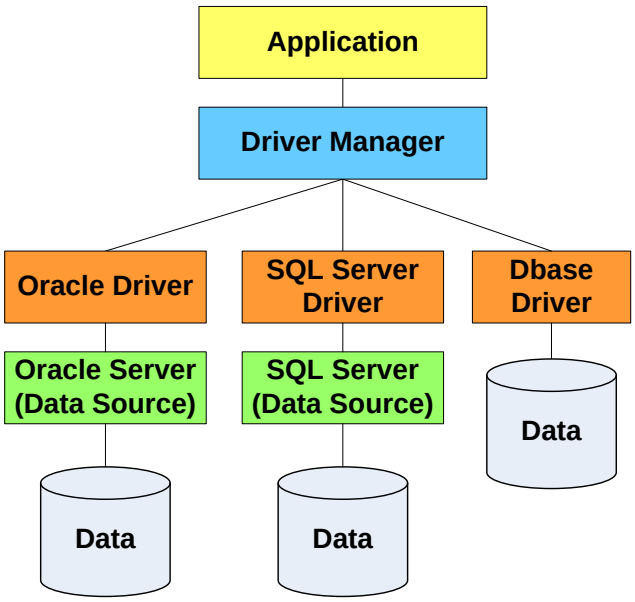

Figure 3.6: ODBC Components

OLE DB

In 1996 Microsoft developed a new connectivity standard by the name of Object Linking and Embedding, Database (OLE DB). It was designed to be a higher level replacement for ODBC with the goal of providing applications uniform access to data stored in both DBMS's and non database containers [13] (such as spreadsheets, ISAM systems [83], and file systems). OLE DB provides applications with the advantages of database technology without having to transfer data from its original source to a database [14]. OLE DB is a set of Microsoft's Component Object Model (COM) [22, 114] interfaces that encapsulate reusable portions of DBMS functionality [25]. Figure 3.7 shows a component architecture using OLE DB interfaces.

OLE DB functional areas include data access and updates, query processing, catalog information, notifications, transactions, security, and remote data access. By defining a uniform set of interfaces to access data, OLE DB components not only contribute to uniform data access among diverse information sources but also help reduce the application's complexity

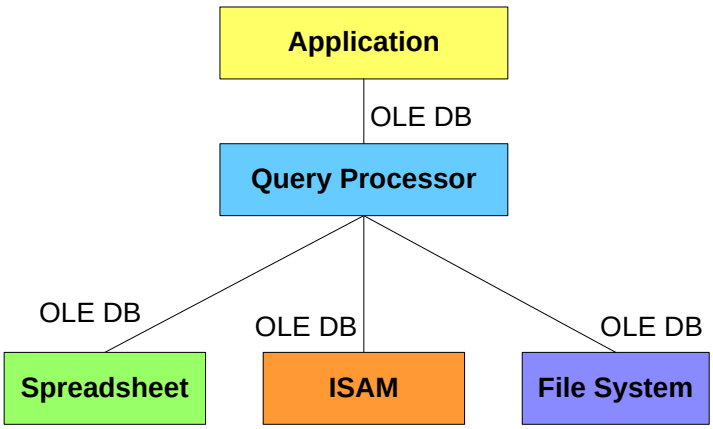

Figure 3.7: A component DBMS architecture using OLE DB interfaces

by enabling developers to use only the DBMS functionality they need [23]. OLE DB offers two programming object models, the "Rowset Programming and Object Model" and the "Binder Programming and Object Model". The rowset model is the traditional DBMS access model, where all information is expected to be contained in rectangular tables where every row represents a data entry [26]. The binder model is designed for cases where data does not fit well into a rectangular table (where entries do not all have the same set of fields). In this case, the binder model associates a Uniform Resource Locater (URL) with an OLE DB object, automatically creating a hierarchy of objects if necessary [71].

JDBC

In 1996 Sun Microsystems released their first draft of the Java Database Connectivity (JDBC) API specification [81]. Since the first final release in January of 1997 [2], the JDBC API has been the industry standard for database-independent connectivity between the Java programming language and a wide range of data sources including SQL databases, spreadsheets, and flat files [79]. The JDBC API provides a call-level API for SQL-based database access [73]. Figure 3.8 shows an overview of the JDBC architecture.

The JDBC API makes it possible to three things [50]:

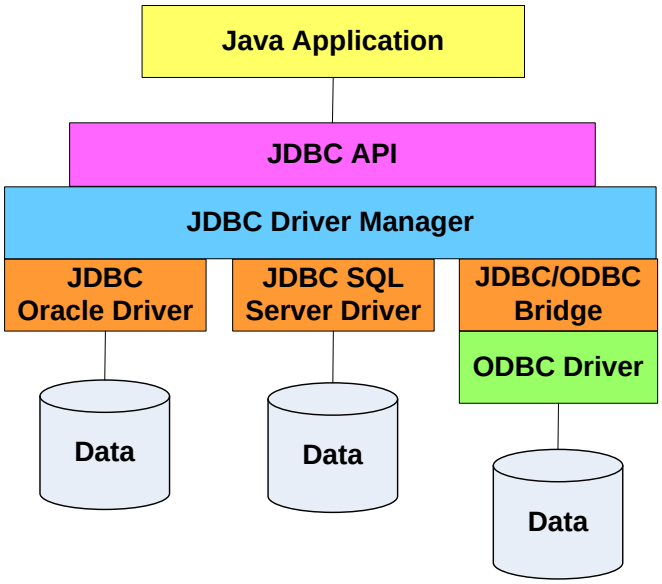

Figure 3.8: JDBC Architecture

- Establish a connection with a database or access any tabular data source
- Send SQL statements
- Process the results

In its initial release, the JDBC API focused on providing a basic call-level interface to SQL databases. The JDBC 2.1 specification and the 2.0 Optional Package specification then broadened the scope of the API to include support for more advanced applications and for the features required by application servers to manage use of the JDBC API on behalf of their applications. The JDBC 3.0 specification aimed to broaden the API by filling in small areas of missing functionality. The JDBC 4.0 specification [2] sets forth two goals: To "improve the Ease-of-Development experience for all developers working with SQL in the Java platform". In light of this, it is worth noting that The JDBC 4.0 Specification draft contained a section called "Ease of Development" that described using Java annotations to represent relational queries, however in the final JDBC 4.0 Specification, the "Ease of Development" section had been removed. The second goal set forth by JDBC 4.0 is to "provide a range of enterprise

level features to expose JDBC to a richer set of tools and APIs to manage JDBC resources" [2].

3.2 Object Oriented Data Mapping

Database connectivity allows programmers to specify queries, run them, and process the results from modern programming languages. However, while the basic model of database connectivity is far better than a vendor specific API, it is not without it's flaws. Information in databases accessed by programming languages through any API (whether standardized or vendor specific) requires that queries be specified as text strings. These queries are significant portions of the program logic yet they are not part of the programming language, and as such offer several disadvantages.

From a programmers standpoint, they must have a knowledge of the database query language, the strings cannot be checked by the compiler for correctness, no IntelliSense support is offered from the IDE, and when an error does occur in the query string it can be incredibly hard to debug. Not only does specifying queries as strings introduce complications, the basic data model between modern programming languages and databases differs greatly. Modern programming languages define information in the form of objects. Databases use rows and columns. Objects have unique identity as each instance is physically different from another. Rows and columns are identified by primary key values. Objects have references that identify and link instances together. Rows and columns are intentionally left distinct requiring related entries to be tied together using foreign keys. Objects stand alone, existing as long as they are still referenced by another object. Rows and columns exist as elements of tables, vanishing as soon as they are deleted.

The solution to these complications is to provide language constructs that map enterprise language features directly the database, and therefore allow a middleware system to generate all of the queries necessary, thereby automating an error prone task. This removes the

opportunity for programmers to make basic string errors (e.g. forgetting a whitespace, omitting a comma, or using the wrong type of quotation mark), and opens the door to the possibility of compile time checks for database queries. Two related but distinct technologies exist that do this very well. The first is a broad range of technologies known as orthogonal persistence, and the second is Microsoft's LINQ project.

3.2.1 Orthogonal Persistence

The first step in understanding "orthogonal persistence" is to define the meaning of the term, which in this case is the result of two root words "orthogonal" and "persistence". Thus "orthogonal" refers to independence between individual concerns in a system, and "persistence" refers to the characteristic of data to outlive the execution of the program that created it. The overarching idea behind orthogonal persistence is to make data access in a program appear to the programmer as if they are simply working with data in main memory [7].

For a more precise definition, orthogonal persistence systems provide an abstraction of permanent data storage that hides the underlying storage hierarchy of the hardware platform. This abstraction is achieved by binding a programming language to an object store, such that persistent objects are automatically cached in volatile memory for manipulation by applications and updates propagated back to stable storage in a fault-tolerant manner to guard against crashes. The resulting persistent programming language and object store together preserve object identity: every object has a unique persistent identifier, objects can refer to other objects, forming graph structures, and they can be modified, where the modifications are visible in future accesses using the same unique object identifier [48].

Atkinson and Morrison [9] state three design principles for designing persistent programming languages:

- Persistence independence: the language should allow the programmer to write code

independently of the persistence (or potential persistence) of the data that code manipulates. From the programmers perspective access to persistent objects is transparent, with no need to write explicit code to transfer objects between stable and volatile storage.

- Data type orthogonality: persistence should be a property independent of type, Thus, an objects type should not dictate its longevity.
- Persistence designation: the way in which persistent objects are identified should be orthogonal to all other elements of discourse in the language. Neither the method nor scope of its allocation, nor the type system (e.g., the class inheritance hierarchy), should affect an objects longevity.

In practicality data is automatically queried from and stored to a persistent data store such as a database or file system without the programmer specifying query strings as part of the program execution (i.e. no JDBC interface calls are explicitly made from the programmer's source code). The automatic mapping is done through the use of standard class objects that in the Java world are commonly called "Plain Old Java Objects" (POJOs) and meta-data information (XML mapping files or Annotations) that provides a middleware system (the orthogonal persistence implementation) with information about how the data in the POJO is related to data in the persistent data store [64]; figure 3.9 gives a high level illustration of the system architecture. Benefits of using orthogonal persistence in enterprise applications include:

- Programs such as procedures and modules can be represented by first class values which reside in the persistent store.
- The persistence mechanism provides incremental loading.
- The persistence mechanism verifies type correctness of linking.

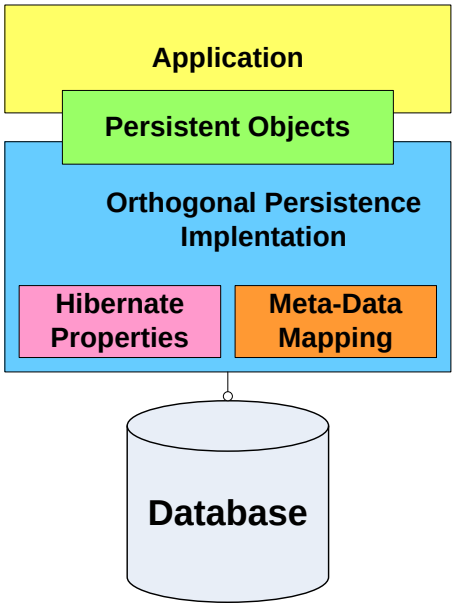

Figure 3.9: Orthogonal Persistence Architecture

- The persistence mechanism supports incremental program construction and replacement.
- The persistence mechanism provides program and data library management.
- Provides another layer of abstraction between the programmer and underlying system, so *should* completely eliminate any DB engine dependence.
- Provides better security by eliminating user written SQL statements.

The concept of orthogonal persistence can be traced to the development of orthogonal persistent programming languages such as PS-Algol [5, 6, 4] in 1981 and Napier88 [30, 69] in 1989. Significant contributions have also been made through extensions to existing languages such as Smalltalk [46, 47, 57, 58, 97] and Java [8, 7, 54, 55]. Major orthogonal persistence implementations currently in use include JBoss's Hibernate [45] for both Java and .NET, Sun's Java Persistence API [49], and Microsoft's LINQ project [70, 27]

Currently Java Hibernate is the most widely developed and available orthogonal persistence

```
@Entity
@Table(name = "MESSAGES")
public class Message {

        @Id @GeneratedValue
        @Column(name = "MESSAGE_ID")
        private Long id;

        @Column(name = "MESSAGE_TEXT")
        private String text;

        //Getter and Setter Methods Below
        ...
}
```

```
EntityManager newEm = emf.createEntityManager();
EntityTransaction newTx = newEm.getTransaction();
newTx.begin();

List messages = newEm.createQuery("select m from
            Message m order by m.text asc").getResultList();

System.out.println( messages.size() + " message(s) found" );

for (Object m : messages) {
        Message loadedMsg = (Message) m;
        System.out.println(loadedMsg.getText());
}

newTx.commit();
newEm.close();
```

Figure 3.10: Java Hibernate Example

solution available for Java. It is a component of the JBoss Enterprise Middleware System [53]. Java Hibernate requires the user to provide an XML mapping file for specifying how the application will communicate with the database, and also provides the option to use either an XML mapping file or annotated classes to map Java POJOs to the database. Figure 3.10 shows an example of a Java Hibernate program selecting two fields ("MESSAGE_ID" and "MESSAGE_TEXT") from the table "MESSAGES" using annotations [12].

3.2.2 Microsoft LINQ

Microsoft decided that existing database API's needed to be taken a step further, so they created the ".NET Language-INtegrated Query (LINQ)" project. The purpose of the LINQ project is to add general-purpose query facilities to the .NET Framework that apply to all types of information, not just relational and XML data [24].

The term "language-integrated query" is used to indicate that query is a built-in feature of the developer's primary programming language (e.g. Visual C#, or Visual Basic). The benefits of integrating queries into the programmer's primary programing language include the incorporation of meta-data into the source code, compile-time syntax checking, static

```
[Table(Name="Customers")]
public class Customer
{
        [Column(Id=true)]
        public string CustomerID;

        [Column]
        public string City;
}
```

```
// DataContext takes a connection string
DataContext db = new DataContext("c:\\northwind\\northwnd.mdf");

// Get a typed table to run queries
Table<Customer> Customers = db.GetTable<Customer>();

// Query for customers from London
var q =
        from c in Customers
        where c.City == "London"
        select c;

foreach (var cust in q)
        Console.WriteLine("id = {0}, City = {1}",
                cust.CustomerID, cust.City);
```

Figure 3.11: Sample C# DLinq Application

typing, and IntelliSense. The ability to statically type, gain IntelliSense support, and have compile-time syntax checking on declarative queries does not currently exist in any other technology that is compatible with a broad range of standardized databases. Language-integrated queries also allow a single general purpose declarative query framework to be applied to in-memory information as well as information from external data sources sources.

DLinq [1] is the LINQ component for managing relational data as objects without giving up the ability to query. It does this by translating the language integrated queries into SQL for execution by the database and then translating the query results back into user defined objects. The users application is then free to manipulate the objects while DLinq tracks the changes automatically. This integration provides strong typing over relational data while retaining the expressive power of the relational model and the performance of query evaluation directly in the underlying store. Figure 3.11 shows an example DLinq application that queries all customers whose city is “London” using a language integrated query.

DLinq provides all of the features of orthogonal persistence implementations, plus the support of an Integrated Development Environment (IDE). By integrating the query language features into the .NET language, the programmer has access to IntelliSense for objects in the

database, syntax checking at compile time, and the ability to statically type data queried from the database. This combination of features is made possible by the fact that Microsoft owns the languages they are providing these features to, and as such are able to include them into the specification and compilers.

3.3 Applicability of Existing Technologies

Now that a clear understanding of the current state of the art related to bridging modern object oriented languages has been obtained, it is important to evaluate their applicability to creating the DJ system. Three major technologies discussed above will be analyzed: database connectivity, specifically JDBC because it is Java's connectivity standard, Java Hibernate because it is a commercial grade orthogonal persistence implementation, and Microsoft LINQ (DLinq for databases) because it currently offers features not available in any other system. The question that must be answered is "can this technology be modified to work or simply used as-is?"

3.3.1 JDBC

Unfortunately the JDBC standard requires the database underlying the connectivity layer be an SQL based database. Thus it is possible to learn from the existing implementation's goals, accomplishments, and lessons learned when developing it, but a JDBC adapter cannot be developed for LogicBlox based on the JDBC standard. Furthermore, the LogicBlox technology is a relatively new technology, and the resources required to develop such an adapter at this point would not be a feasible or wise investment of time.

Were the JDBC standard written in such a way that an adapter could be developed to work with the LogicBlox database, it would not satisfy the goal set forward of removing the burden of learning Datalog from enterprise programmers since JDBC still requires that

query language strings be passed in. Also, as was discussed in the "Object Oriented Data Mapping" section, this solution still places the burden of translating data from its storage format in the database to the object oriented language on the programmer. DJ mus provide an intuitive way to map column-oriented data into Java objects.

3.3.2 Java Hibernate

The next technology to consider is orthogonal persistence, and more specifically Java Hibernate. The question specifically becomes can Java Hibernate be modified to work with LogicBlox. Even though Hibernate sits on top of a JDBC driver, the project is open source, so in theory it could be re-written to leverage the LogicBlox libraries or a JDBC like driver that sits on top of the LogicBlox API. This however is unfeasible due to the constraints imposed by the fact that Java Hibernate was designed to model relational data in an object oriented framework. LogicBlox does not store data by rows like a relational database, but instead stores data in a column-oriented fashion. Hibernate thus assigns a class to be representative of a table, and specific members of that class to each represent specific fields in the modeled table. However to simply modify hibernate to work with the LogicBlox column oriented world, each predicate would require it's own class with a single member variable. This quickly becomes impractical, and the need for a better way to encapsulate data from multiple predicates inside a single container quickly manifests itself.

3.3.3 Microsoft LINQ

The Microsoft LINQ project, and more specifically the DLinq project offer some very interesting insights into what can be accomplished in a development environment with full database query language integration. Unfortunately the technology is inapplicable for the task of creating a LogicBlox data bridge. Microsoft owns the languages that are are included in the Linq project (C# and Visual Basic), therefore capabilities can be added to the lan-

guage specification with full IDE and compiler support. In the case of the LogicBlox project, since Java is used, and the resulting code must be compilable with standard Java compilers, and executable on the standard Java Virtual Machine, adding new features to the language is not possible in order to provide support for the project, so standard Java constructs must be used.

Chapter 4

DJ: A Java to Deductive Database Bridge

This chapter overviews the DJ system, which is a middleware bridge for providing access to the LogicBlox database for Java programmers. The DJ system resulted from the collaboration of academic research and an industry need to ease the integration of enterprise software systems with LogicBox's column oriented deductive database. A presentation of different design approaches that were looked at and tested is given, then an overview of the final DJ system is presented, followed by an analysis of the work done.

4.1 Solution Approaches

When considering the best approach for designing the DJ system, several different approaches and considerations were taken into account. Some of these were evaluated at the conceptual level, while others were tested in a prototype environment and presented to LogicBlox for evaluation from an industry standpoint.

4.1.1 JDBC Like Approach

There are quite a few variations to the approaches that can be taken to solve this problem. The most simple of these is to create a JDBC like connectivity bridge that accepts Datalog queries as strings, and then uses a middleware application to communicate with the database. This is not incredibly desirable, because the burden is still on the programmer to learn Datalog. The problem presented with the approach of representing query strings as string objects still exists, an as such the inability for compile time checking, likelihood for basic errors, and potential difficulty in debugging.

4.1.2 Meta Data Types

For this project two types of meta-data are important to consider. Annotations in the source code itself, and an external file such as XML. Annotations offer the benefit of allowing programmers to see the relationship of the meta-data to the code while they are programming. Annotations however offer the disadvantage of locking the information in at compile time. Anything not known at compile time cannot be specified via annotation, and thus cannot be represented in code. XML files offer the inverse of annotations. XML does not allow the programmer to see the relationship of the meta-data to their source code during development. However XML files do not require that all information be specified at compile time, and therefore make changes easier to deal with. It was the decision of this project to use annotations, due to the complexity of the problem, it would seem that flexibility to specify mapping information after compile time is not a useful asset, since no data container would be written to hold the mapped data, and the maintenance issues inherit with maintaining code that does not self-comment by displaying the mapping meta-data in-line with it's data container will become prohibitive as new developers unfamiliar with the underlying technologies and database implementations are added to the project.

4.1.3 Meta-Data Datalog Queries

The second approach that can be taken is to simply allow basic description via annotations such as Java Hibernate on objects in a class or interface that use Datalog to describe the desired query results. This is not ideal because Datalog strings are still used, but at the same time, since data is specified in a meta-data fashion at compile time as a constant string, the potential exists for IDE integration (i.e. Eclipse plugin) in order to verify syntax legality at compile time. Also, since the programmer has specified all of the data at compile time in an annotated class or interface, in the event of errors there is not a hard time tracing through the program looking for the cause, so the Datalog strings can be statically evaluated by a trained programmer and basic errors easily caught. This solution still does not remove the burden of learning the specific query language from the programmer. This solution also imposes the problem that queries must be specified at compile time, and runtime information cannot be used to create queries, thus limiting the use to bridging the class of queries that have no runtime conditions present.

4.1.4 Database Representative Java Classes

The next approach that can be considered is that of using a large library of classes that collect information from the programmer and then create query objects that interact with the middleware system on behalf of the programmer. This option does allow information to be set at runtime, and therefore offers flexibility. It also offers the benefit of abstracting the programmer from the burden of learning Datalog, they simply need to learn the provided Java classes, and what information to pass in for the creation of dynamic queries. This solution is not without it's flaws though, in that an incredibly large burden of code writing is placed on the programmer, and even very simple queries which may be expressible in one line of Datalog will result in the creation of many lines of Java. The more lines of Java code that are introduced, the more likley a programmer is to make a simple mistake, and the more costly software maintenance becomes. This approach also forces the Java classes to closely

model the Database object structure, and therefore reduces the level of abstraction, and makes re-use with new engines potentially unfeasible. This also adds to the potential that database API changes by the vendor will specifically affect customer code, thereby offsetting one of the advantages of a middleware system.

4.1.5 Classes versus Interfaces

An important design decision that had to be made was what Java language construct was going to be used as the "container" to hold the data in. Two options exist, that of a class, or Plain Old Java Object (POJO), or an interface. Classes offer the advantage that they allow member variables to be dynamic, as opposed to an interface in which member variables must be static final. With classes, the programmer can provide code for their own methods that operate on data directly inside of the class. However, classes have the disadvantage that for a middleware system, point-cuts must be used by the middleware system to intercept method calls. Also, good object oriented design principles say that member variables inside a class should be private, and only accessible by getter and setter methods. An interface on the other hand can utilize the Java construct of creating dynamic proxies, and no point-cutting needs to occur at runtime making development of the middleware system much easier. Method calls can be used to access data that does not require a member variable to be provided in interface definition. For the remainder of this section, the assumption will be made that interfaces have been selected for use in the LogicBlox–Java Bridge.

4.1.6 Meta-Data Bindings

The final approach to be considered is the concept of "binding" interface methods to specific entities in the database. In this approach a provided "Bind" annotation is used on the interface as a whole to create the concept of an "entity" or logical grouping of data, and then individual methods of the interface to map them directly a predicate. Since in the

LogicBlox data model predicates only store a single value, it can be assumed that the return type of the method maps directly to the stored value type of the predicate, and when the getter function is called it will return a value for that predicate, and when a setter method is called it will set the value for that predicate. This solution eliminates the need for the programmer to understand Datalog, but does require a keen understanding of the underlying database. However this is no different than if a programmer needs to actually write Datalog query strings, since a query string cannot be written without explicit knowledge of of the underlying database. Also, the need for knowledge about an underlying database can be offset by the provision of graphical tools that allow a programmer to browse a database and then generate the annotated interface from an existing database. The dilemma of burdening the programmer with more Java code for ease of understanding is removed since a simple annotated interface is used to represent the data, while the drawback of only being able to specify queries at runtime is eliminated. This is eliminated because no queries are being used. The mappings point directly into data stores, so as the data in the stores is changed, it will propagate back into the program.

4.2 DJ Implementation

The DJ system takes the approach of representing queries and the database in a very similar manner to Java hibernate. The system binds Java interfaces to fields in the database through annotations which provide the mapping at compile time. DJ uses a middleware system that creates an in-memory dynamic proxy at runtime to communicate with the database, thereby abstracting all database API calls from the programmer. Not only does DJ abstract database API calls from the programmer, but it can completely eliminate the need for a programmer to learn and write queries in Datalog, the database language of LogicBlox.

The implementation of DJ is best demonstrated through a running example. Consider the problem of implementing a system that records orders for a retailer. The database structure

Table 4.1: Item Database Visualization

Item		
SerialNumber	Name	Price

Table 4.2: LineItem Database Visualization

Item		
Item.SerialNumber	ItemQuantity	LineItemPrice = (Item.Price * ItemQuantity)

with relevant entries holds a collection of data for individual items, specifically an item's name, serial number, and price as visualized in table 4.1. An "order" then consists of a collection of "line items", where a line item is the quantity of an item ordered. The line item records in the database will consist of the item's serial number, the quantity of that item ordered, and the line item price which equals the quantity ordered times the price: table 4.2 illustrates a line item database entry. Finally an order record consists of the customer that placed the order, the date the order was placed, the total collection of line items in the order, and the total order price is the sum of the line item prices: table 4.3 gives a visual representation of an order record and table 4.4 shows the customer's data representation.

It is important to clarify that this example is not for the purposes of demonstrating the uses or benefits of LogicBlox's column oriented deductive database; the example was chosen to provide a running example to help visualize the concepts and contributions of the DJ system. Both deductive and column oriented databases are optimally used in situations involving many many records, and incredibly complex data relationships, which make it impossible to provide an example that both simply and efficiently helps to demonstrate the DJ system and can claim to be a problem for which a column oriented deductive database is the optimal DBMS. For a better understanding of deductive databases please see chapter 1, and for an overview of column oriented databases please see chapter 2.

Table 4.3: Order Database Visualization

Order			
Customer	Date	Collection<LineItem>	TotalPrice = Sum(LineItem.Price)

Table 4.4: Customer Database Visualization

Customer	
Name	idNumber

One of the unique contributions of the DJ system is that it uses annotated Java interfaces or "Plain Old Java Interfaces" (POJIs) to represent data stored in the database as opposed to annotated classes or POJOs. There are several annotations that are used in annotated interfaces, but most notably the *@Bind* annotation which binds methods in an interface to underlying records in the database. This feature of DJ is best understood by looking at sample code that creates working Java code using the order system example.

4.2.1 Example in Datalog

When considering how to write Java code in the DJ system, it is important to understand exactly what the annotated interfaces are abstracting from the programmer. Therefore the first problem that DJ solves in this case is representing the database in a way that enterprise programmers can intuitively understand. To gain a better understanding of the complications imposed by the technology on enterprise programmers unfamiliar with LogicBlox technology consider the Datalog code required to create the database for the previous example.

```
Customer:customer(c) ->.
Customer:name(c;s) -> Customer:customer(c), string(s).
Customer:idNumber(c;s) -> Customer:customer(c), string(s).

Item:item(i) ->.
Item:serialNumber(i;sn) -> Item:item(i), string(sn).
Item:name(i;s) -> Item:item(i), string(s).
Item:price(i;p) -> Item:item(i), float[32](p).

LineItem:lineItem(li) ->.
LineItem:item(li;i) -> LineItem:lineItem(li), Item:item(i).
LineItem:quantity(li;n) -> LineItem:lineItem(li), int[32](n).
LineItem:price(li;p) -> LineItem:lineItem(li), float[32](p).
```

```
Order:order(o) ->.
Order:customer(o;c) -> Order:order(o), Customer:customer(c).
Order:date(o;d) -> Order:order(o), datetime(d).
Order:lineItems(o,li) -> Order:order(o), LineItem:lineItem(li).
Order:price(o;p) -> Order:order(o), float[32](p).
```

Presented with the above task of learning to model data in a way that fits logically into the declarative style of programming shown above is a very daunting task for programmers familiar with traditional RDMBS technology. Each predicate is representative of a single column of data, and therefore a programmer will have to model the data in a table wise fashion, and then attempt to logically split it into columns, and then code the correct Datalog declarations to create the database. Enter DJ: by modeling the data as Java interfaces, the programmer will be able to automatically deploy their database from their Java code. The previous lines of Datalog are not needed, and are replaced by DJ's annotated interfaces.

To gain a better understanding of the DJ annotations, the following sections will replace a block of Datalog representing specific functionality with the equivalent Java interface. The section will then detail features of the DJ Annotations represented in that portion of the example.

4.2.2 Basic "Bind" Functionality

The first block of Datalog represents the "Client" functionality portion of the database. The client functionality portion is very basic, simply containing a "name" and "idNumber" for that client. For this example, DJ replaces the Datalog block:

```
Customer:customer(c) ->.
Customer:name(c;s) -> Customer:customer(c), string(s).
Customer:idNumber(c;s) -> Customer:customer(c), string(s).
```

with the annotated Java interface:

```
@Bind(namespace=``Customer'', entity=``Customer'')
public interface Customer
{
        @Bind(predicate=``name'')
        String name();

        @Bind(predicate=``idNumber'')
        String idNumber();
}
```

What is happening here is that the interface *Customer* is a full representation of the underlying database, and as such the database can be created from the interface. The *@Bind* annotation that annotates the interface as a whole indicates a "namespace" value, which is used by LogicBlox in their data modeling. The namespace corresponds to the name of the predicate that is to the left of the last colon. So in this case, the predicate "Customer:name" has the namespace of "Customer", and the predicate name of "name". The entity entry is used to indicate that the collection of values inside the interface are logically related, and in the relational world would most likely translate into a single table. It is important to note that if the "namespace" value is omitted from the *@Bind* annotation, then a blank namespace will be assumed unless otherwise indicated by a "namespace" value on an individual predicate. For example, if the "namespace" value were to be omitted from the interface's annotation, meaning that the existing annotation:

```
@Bind(namespace=``Customer'', entity=``Customer'')
```

would be replaced by:

```
@Bind(entity=``Customer'')
```

The DJ system would then look for the the "name" and "idNumber" predicates in the default namespace, so it would simply attempt to query "name" and "idNumber" rather than "Customer:name" and "Customer:idNumber". However, were the interface to be re-annotated in the following manner everything would continue to function correctly.

```
@Bind(entity=``Customer'')
public interface Customer
{
        @Bind(namespace=``Customer'', predicate=``name'')
        String name();

        @Bind(namespace=``Customer'', predicate=``idNumber'')
        String idNumber();
}
```

4.2.3 More Advanced "Bind" Functionality

The next block of Datalog functionality presented in the Datalog code is that for an item's data representation. This section will allow for an overview of some of the features of the annotations that facilitate an easier experience for the developer. Hence the Datalog code representing an individual item's data:

```
Item:item(i) ->.
Item:serialNumber(i;sn) -> Item:item(i), string(sn).
Item:name(i;s) -> Item:item(i), string(s).
Item:price(i;p) -> Item:item(i), float[32](p).
```

is replaced with the annotated Java interface.

```
@Bind(namespace=``Item'')
public interface Item
{
        @Bind
        String serialNumber();

        @Bind
        String name();

        @Bind
        float price();
}
```

In this case, it is interesting to note that the "entity" value of the *@Bind* annotation for the interface level has been removed, as well as the individual "predicate" values for the individual method *@Bind* annotations. This is due to the fact that DJ wishes to reduce the burden on a programmer as much as possible. Therefore DJ makes the logical assumption that if an "entity" value is not specified for an interface, than the entity value is simply the same as the name of the interface. Likewise, if the "predicate" value is omitted from an individual method's annotation, then DJ makes the assumption that the name of the predicate is the same as the name of that method. The *@Bind* annotation is however not optional, as DJ will not assume that a method or interface is bound to a database. This allows for the programmer to provide methods and fields not corresponding to the database in an interface that is bound to a database.

4.2.4 Getter – Setter Functionality

Currently all of the Java code that has been presented is read only code for the user. Therefore the next feature of DJ annotated interfaces that is discussed is that of both getting and setting data field values. This will be done through the example of the line item's data representation. Thus, the Datalog code for a "line item":

```
LineItem:lineItem(li) ->.
LineItem:item(li,i) -> LineItem:lineItem(li), Item:item(i).
LineItem:quantity(li;n) -> LineItem:lineItem(li), int[32](n).
LineItem:price(li;p) -> LineItem:lineItem(li), float[32](p).
```

becomes the annotated interface:

```
@Bind(namespace=``LineItem'')
public interface LineItem
{
        @Bind
        Item item();
```

```
        @Bind
        int quantity();

        @Bind
        float price();
}
```

However, this is an interface, and therefore all data representation is done through methods, not fields. All methods up to this point have simply returned a value, and therefore are "read-only". The approach that DJ takes to this, is to require the programmer to annotate another method with a *void* return type, that takes the value's type for the predicate as a parameter. So the previous interface with "get" and "set" functionality would appear as follows:

```
@Bind(namespace=``LineItem'')
public interface LineItem
{
        @Bind
        Item item();
        @Bind
        void item(Item i);

        @Bind
        int quantity();
        @Bind
        void quantity(int i);

        @Bind
        float price();
        @Bind
        void price(float f);
}
```

Now the user has both the ability to write data back to the database as well as read data. This method may appear to be a bit cumbersome from an "ease of use" perspective, but it is important to remember that related technologies such as Java Hibernate provide examples that use private fields for data representation, and then use getter and setter methods to access those field values. Other features of the DJ system implementation to help alleviate

this burden of two methods for a single field will be discussed and analyzed in later sections.

Another interesting feature presented in the current *LineItem* interface worth noting is that annotated interfaces are not limited to native Java types. Here an object of type *Item* (another annotated interface) is returned to the *LineItem* interface. DJ automatically handles this data translation for the user, and the user will receive an object of type *Item* when the "item()" method is called.

4.2.5 Set Handling

The last key feature of DJ annotated interfaces that has not been presented yet is the ability to handle "parent-child" relationships in the form of a set. Meaning that the annotated interface is treated as the parent and then a collection of items is the result of a query, and they are all children of the parent. This is the case for the order data functionality. An order can have a collection of line items associated with it, but each line item has only one order associated with it. The Datalog code representing this functionality is:

```
Order:order(o) ->.
Order:customer(o;c) -> Order:order(o), Customer:customer(c).
Order:date(o;d) -> Order:order(o), datetime(d).
Order:lineItems(o,li) -> Order:order(o), LineItem:lineItem(li).
Order:price(o;p) -> Order:order(o), float[32](p).
```

which becomes the following interface:

```
@Bind(namespace=``Order'')
public interface Order
{
        @Bind
        Date date();

        @Bind
        Customer customer();

        @Bind
```

```
        float price();

        @Bind
        ArrayList<LineItem> lineItems();
}
```

In this case the "lineItems()" method represents the collection of the order's line items or children. Currently the DJ system uses the presence of an *ArrayList* to determine that this will be a parent child relationship. The fact that the collection (via *ArrayList*) is an element of the interface indicates that the interface is the parent. The type provided to the *ArrayList* (in this case *LineItem*) indicates that the children of the interface are of that provided type. When the method "lineItems()" is called the *ArrayList* filled with the parent's children will be returned, and the children can be iterated through using standard Java conventions for an *ArrayList*.

4.2.6 Business Rule Support

The last piece of logic missing from the orders example is that of calculating the price values for the line items and the orders. As was discussed in the LogicBlox overview the ability to place rules on a field and calculate its value from other predicate values comes with a deductive database. Therefore, why force the programmer to do the calculations in their Java code? The Datalog to do this was omitted from the Datalog example since it is orthogonal to the declaration of the database and the functionality can be added later, or may not even be used if the developer prefers to handle the calculations in the enterprise application. The Datalog code for calculating both the *LineItem* "price", and the *Order* "price" is as follows:

```
LineItem:price(li;p) <- LineItem:item(li;i), LineItem:quantity(li;q),
        Item:price(i;ip), p = q * ip.

Order:price(o;tot) <- agg<<tot=total(p)>> LineItem:price(li;p),
        Order:lineItems(o,li).
```

In the DJ model, these will be placed inside a *@Calc* annotation that annotates the field they specify the calculation rule for as follows:

```
@Bind(namespace=``LineItem'')
public interface LineItem
{
        @Bind
        Item item();

        @Bind
        int quantity();

        @Bind
        @Calc(formula=``item.price * quantity'')
        float price();
}

@Bind(namespace=``Order'')
public interface Order
{
        @Bind
        Date date();

        @Bind
        Customer customer();

        @Bind
        @Calc(formula=``total(lineItems.price)'')
        float price();

        @Bind
        ArrayList<LineItem> lineItems();

}
```

In this case, instead of forcing the programmer to write the complex Datalog rules that were shown above, DJ allows the programmer to intuitively specify the calculation rules for a field value using a combination of object oriented principles and spreadsheet like formulas. In the formula any fields in the interface are provided by their name, and if they are an object that represents another data type (such as in the *LineItem* example, "item" represents the

interface *Item* so the object oriented "dot" notation is used to provide the specific field value "price"). The basic aggregation method "total" coupled with a value such as "lineItems" whose type is a collection, can determine through reflection that it will sum the individual values returned from that collection of objects. DJ will deploy these business calculation rules at the same time the database is deployed from the interfaces as detailed in the following section.

4.2.7 Using the Annotated Interfaces

After all of the annotated interfaces have been created, the last two issues that remain are creating the database from the annotated interfaces, and using the annotated interfaces to for data management. DJ provides factory methods for deploying the database, and a factory container for holding a collection of records represented by the annotated interfaces.

DJ handles database deployment by accepting the interface's *class* object as a parameter, and is then able to use reflection to generate all of the required underlying Datalog. This benefits the developer by allowing them to write an interface that will be used for data access while removing the burden of declaring their database definition in a paradigm they are not familiar with. Deploying the database in DJ would be done as follows:

```
DJFactory.DeployDatabase(Customer.class);
DJFactory.DeployDatabase(Item.class);
DJFactory.DeployDatabase(LineItem.class);
DJFactory.DeployDatabase(Order.class);
```

It is important to note however that the order of deployment is important in DJ's case. Because dependencies exist on previous declarations, an item which has a dependency on it in a later declaration must be declared first. For example, the current order management system *LineItem* is dependent on *Item* so the *Item* interface must be deployed to the database prior to the *LineItem* interface or an error will occur. However if no dependencies exist, then the order of declaration does not matter. For example *Item* has no dependency on *Customer*,

so it would not cause a problem to deploy *Item* prior to deploying *Customer*. This constraint exists in all data management systems, and should be very intuitive to any developer familiar with deploying databases.

Accessing the data is as simple if not simpler than creating a *ResultSet* with JDBC. DJ provides a container called a *DataSet* which is a Java generic and accepts the interface representing the desired data as a parameter. The DJ factory method "intializePOJI()" is then called accepting the interface's *class* object as a parameter, and the corresponding *DataSet* is then returned. An example of this functionality is shown below:

```
DataSet<Order> orders = DJFactory.initializePOJI(Order.class);

System.out.println(orders.size() + `` orders in the system'');
for (Order o : orders) {
        System.out.println(``Date: '' + o.date());
        System.out.println(``Customer Name: '' + o.customer().name());
        System.out.println(``Total Price: '' + o.price());
        System.out.println(``LineItem.price, LineItem.quantity'');
        for (LineItem li : orders.lineItems())
                System.out.println(li.price() + ``, '' + li.quantity());
}
```

The above Java code queries the entire collection of records in the system, and then iterates through them printing pertinent information. The provided DJ *DataSet* supports the use of Java Java 5 style iterators as shown above.

4.2.8 Generating Annotated Interfaces

The last key piece of functionality provided by DJ is the ability for programmers to generate annotated Java interfaces from an existing database. A Java application was created that allows a programmer to browse a database namespace, and then select which fields they would like to include in their annotated interface. If a particular data field is not needed in an application then there is no need to bind the interface to it. Figure 4.1 shows a screen

shot of the code generation tool.

In the current development status of DJ, the code generation tool was left in a primitive form due to the "proof-of concept" nature of the tool. The user is left with the responsibility of passing in the database path as a command-line parameter, manually typing the namespace in the "Namespace" text box, and specifying the output directory via the "File Location" text box. It would however require minimal effort to add the ability for a user to browse databases, namespaces, and select the output folder for the generated files via a graphical interface. Once the "Generate" button is clicked, an interface will be generated with the name provided in the "Entity Name" field. An entity name is required in the case of code generation due to the fact that the system should not presume what the programmer wants the entity name to be; where in the case of an annotated interface, if the entity name is omitted, then the system simply assumes the entity name directly corresponds to the predicate name. Clicking the "Generate" button in figure 4.1 will produce the annotated interface *Order* that was presented in the "Set Handling" sub-section of the "DJ Implementation" section.

4.3 Evaluation

One of the unique things about the DJ system is that it was developed as an academic research project in collaboration with an industrial sponsor to solve a real-world challenge . This unique combination allowed for the development and experimentation of many unique ideas and concepts in the system. When setting out to evaluate the implementation and approach of the created system, it is important to recall that the work was done as academic research, and the resulting implementation was designed to be a working prototype that provided a proof of concept implementation for the industrial sponsor. The agreement was made ahead of time that the working prototype would be delivered by the academic researchers to the industrial sponsor, and it would then be the industrial sponsor's task to upgrade the system to be industry production quality.

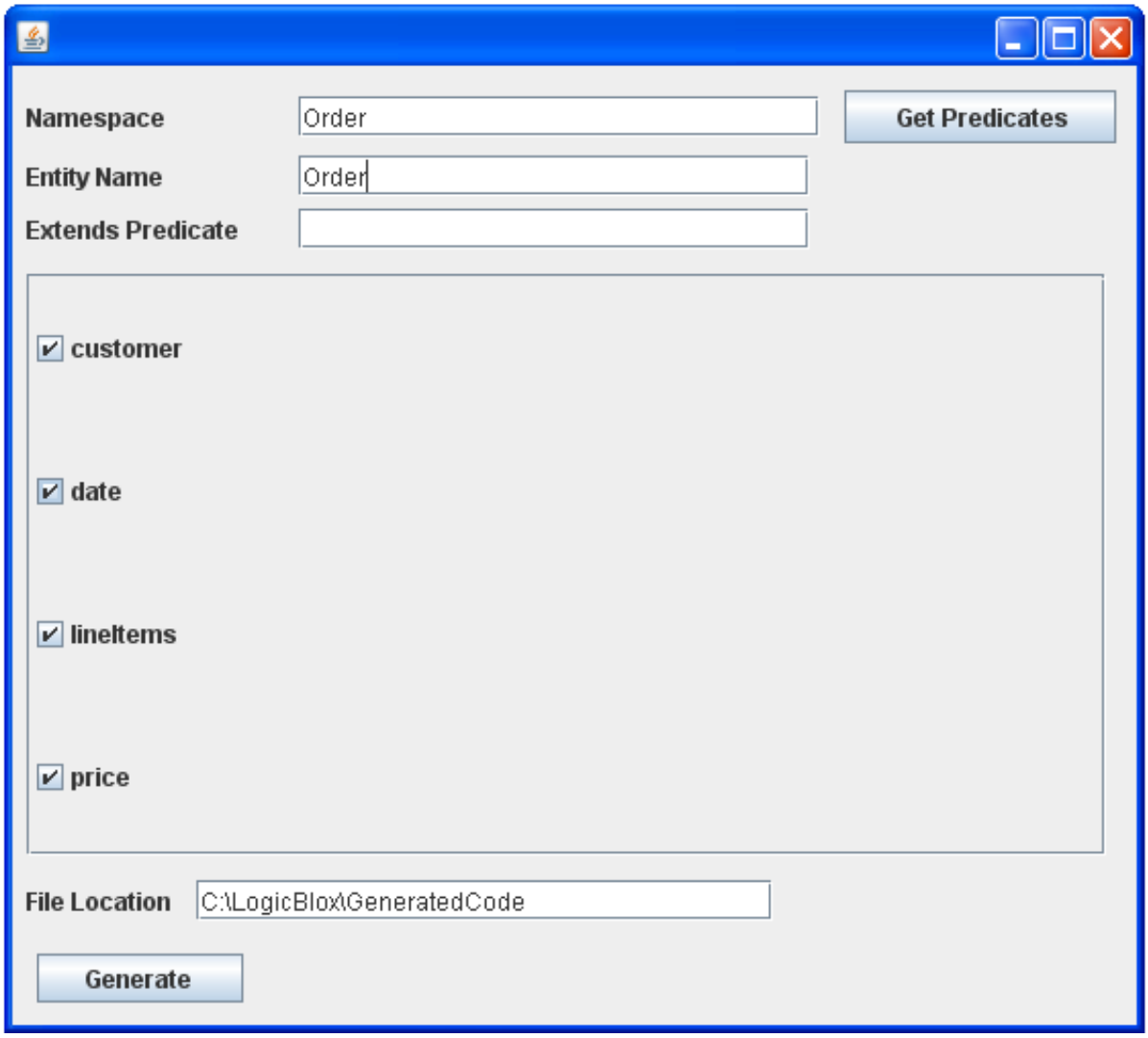

Figure 4.1: Code Generation Tool

The evaluation of the project should be done in light of the goals that were set forth in the chapter introducing the problem. The overall motivation of the project was to provide LogicBlox with a middleware bridge system that would facilitate easy development with their technology, and therefore provide them a marketing edge when attempting to bolster the adoption of their technology. This was done inside the framework of the following explicit goals:

- The system shall use only standard Java language constructs
- The system shall provide a library of Java Annotations available for mapping Java interfaces to data stored in the deductive database
- The system shall include a middleware layer that handles all communication between Java applications and the database.
- Tools shall be provided that allow programmers to generate annotated Java interfaces from an existing database.
- Tools shall be provided that create and deploy a new database (including calculation rules for individual fields) from properly annotated Java code
- The system will allow a programmer with limited knowledge of a deductive database to quickly develop a software system written in Java that uses a deductive database.
- The system shall reduce the maintenance costs of a software system that interacts with a deductive database.

When evaluated in light of these goals, the DJ system is a resounding success. The overall presentation of the system as presented in section 4.2 fully satisfies the requirements. The first stated goal was to use only standard Java language constructs so that any DJ implementation could be used with the standard Java Development Kit (JDK). The DJ implementation does this. It simply uses Java 5 annotations to provide database mapping information to the middleware system.

4.3.1 Annotated Approach

The use of annotated POJIs by DJ reduces the development time and therefore development costs of the middleware system for accessing the column oriented deductive database. This principle can be leveraged in the future as a learning experience for other companies desiring to develop middleware bridge technologies. One of the key challenges that was presented for a startup company is the ability to balance "good-enough" API development (and other "ease of development" features) for a product with a small team of developers struggling to get a quality product to market. Ordinary orthogonal persistence must use point-cutting (done through byte-code engineering) to intercept calls on the POJOs data containers. While this has a lower overhead cost when first creating the objects in memory, it requires much more complex software development at the middleware level. By choosing to go with the annotated POJIs DJ is able to leverage Java dynamic proxies that generate the logic for method calls from the interface at runtime. This makes for much faster development time, and lower software maintenance costs at the middleware level. The use of interfaces with dynamic proxies by the DJ system demonstrates the feasibility for an alternative middleware system that can be developed far more rapidly and requires less knowledge of byte-code engineering.

The Annotation library that was included fully allows the developer to map their Java interfaces to a column oriented database. The use of both annotated getter and setter methods fits into the Java coding convention very logically; since even Java Hibernate uses private fields inside a POJO and then requires getter and setter methods. DJ also helps to alleviate this burden in the case of an existing database by allowing a developer to generate the interfaces with both the getter and setter methods. It also allows for a developer to provide a read-only interface to the database in the case where data should not be modified. Other benefits of the annotation library include the ability to specify calculation rules for a database in an intuitive manner. This functionality is a product of the underlying deductive database, however it is quite possible that the principle of annotating the rule could be used to generate triggers in a traditional RDBMS for calculating field values as well. Thus the

principle of annotating calculation fields could potentially be applied to existing technologies to increase the scope of their functionality.

Drawbacks of the current DJ system of using only annotated interfaces is that it does not allow for complex queries. Databases support the ability to create complex data relationships in memory via their query language, and the results never reside in permanent memory. The DJ system currently restricts data access to an in-memory representation of the database. This is great for basic data access, but prevents complex uses of the database that are often commonplace in industry, and is therefore an area of future work that would need to be addressed to produce a fully functional enterprise quality solution to the problem.

DJ very nicely demonstrates the ability to abstract a programmer from Datalog, as was demonstrated in section 4.2. If a programmer can model their data in an annotated interface, then the DJ system can be used to deploy the database. The section also indirectly demonstrates the middleware abstraction from the LogicBlox database API. This is shown by the fact that only the *DJFactory* methods are called, and the annotated interface class objects are then passed in as a parameter. By including this layer of abstraction between the user and the LogicBlox API, the developer's code is not sensitive to updates in the LogicBlox API. Should a major update be required by LogicBlox to their API, they would simply release a new version of the middleware library, and the client's code would continue to function correctly.

The potential drawbacks of the DJ middleware implementation include performance degradation due to the extra layer of abstraction, and the potential that a developer interested in learning the deductive database technology would have a harder time doing so due to the abstraction away from writing queries as Datalog strings. The DJ system also imposes an extra burden on the LogicBlox vendor to maintain another component of the system. Since clients simply call into the factory methods provided by LogicBlox, it is one more code library that a small company must continue to maintain. However, the marketing benefits of helping clients easily develop tools using their technology on the backend outweigh the

downside of maintaining the code library. In fact LogicBlox has even begun to incorporate components of the DJ system into their own in-house code libraries helping them more easily maintain their product.

4.3.2 Code Generation

The final component of the DJ system not yet discussed is the ability to generate annotated interface code from an existing database. This functionality was demonstrated in section 4.2.8 through the presentation of the provided tool with a graphical interface for browsing namespaces and generating corresponding annotated interfaces. The tool is far from being complete, but does provide an excellent proof of concept and is still useful for generating interfaces. The ability to generate annotated interfaces serves two major purposes. The first and foremost is that it reduces the burden on developers when writing code to interact with the database, and therefore reduces the potential for errors in hand written code. The second benefit of code generation is it allows developers first familiarizing themselves with the technology to better understand the DJ model by creating example interfaces from example databases. This will will help them to visualize how their own data will be represented in the system, and how to write their own annotated interfaces.

The code generation features of DJ lack several key areas of functionality. First of all the tool does not allow a programmer to browser namespaces in the database, it only allows them to browse individual predicates in a single namespace. The tool needs to have functionality to browse namespaces as well as predicates. The second piece of functionality that is lacking is the ability to browser for a database, or at least specify it from the graphical interface. Currently the tool requires the user pass in the database location as a command-line parameter. Next the tool restricts the developer to generating annotated interfaces that represent a single namespace. The functionality should be added to to allow for the inclusion of predicates from multiple namespaces inside a single interface. The DJ system itself supports this by annotating every individual field with the same *@Bind* annotation which allows the

specification of a unique namespace for every method.

A very interesting lesson that was learned while developing the code generation tool of DJ was the ability to generate code that represents a parent-child relationship. The problem that exists when generating code from a database, is that the generator can tell a relationship exists between entities, but cannot tell what sort of relationship it is, meaning is it a "many-to-many" a "one-to-many or a "one-to-one" relationship. In order to support this issue that was raised when researching code generation, LogicBlox added meta-data predicates to their database model for providing this information so code could be correctly generated. This problem is not easy to solve with traditional technologies, as was demonstrated by a test case with JBoss's Hibernate tool set for Eclipse. A relational database was generated from sample annotated classes in the Java Hibernate documentation, a set of annotated Java classes were then generated from the database that was created, and the two sets of annotated classes did not match. This shows that the DJ system is able to address a complex problem that an existing technology cannot due to the generality of the scope in which it operates.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

The development of the DJ system shows that deductive databases and object oriented languages can be successfully bridged using middleware technology written in standard Java. Not only does DJ demonstrate the feasibility of bridging deductive databases with an object oriented language, it also develops and sets forth principles for modeling column oriented data in an object oriented language, since the LogicBlox database is a column oriented deductive database. The partnership between academic research and industry for the development of DJ ultimately resulted in a higher quality and more unique solution. By developing the project as academic research there was not an immediate pressure to produce a solution that worked, and it allowed for the exploration of many different approaches to solving the problem. The problem that developing solutions in industry sometimes poses is that a solution needs to get done, and only existing technologies are used, where in an academic environment there is freedom to explore alternative technologies that may or may not produce a return on the time investment. However academic research is sometimes plagued by the problem that as long as it is unique, it is interesting and therefore development of viable solutions often does not occur. In the case of DJ the quality and usefulness of the system was

kept in check by developing the system with an industrial partner. The industrial sponsor provided great oversight for the work that was done, since the question "is it useful?" was constantly being asked and answered.

The DJ system consisted of four major components:

- A library of Java annotations for mapping interfaces to a database
- A middleware system for communication with the database
- Tools for deploying a database from annotated Java interfaces
- Tools for generating annotated Java interfaces from an existing database

It was the combination of these components that allows the stated objective of "allowing programmer with limited knowledge of a deductive database to quickly develop and easily maintain a software system written in Java that uses a deductive database as its data management system" to be successfully met. The use of dynamic proxies in conjunction with interfaces shows that rapid development of a middleware system can be done without the need for byte-code engineering, therefore making the DJ system a valuable starting point for developing middleware bridges for emerging database technologies. The fact that DJ is able to abstract the programmer from needing direct knowledge of Datalog also shows that the barrier of developing technologies that developers are unfamiliar with can either be reduced or eliminated through successful middleware technologies.

Ultimately the creation of the DJ system was an excellent "proof of concept" for middleware bridges. However DJ was developed as a small academic project with a small industrial sponsor. The work done provides an excellent starting point that can be leveraged to explore the field in far greater depth. A key question that is raised by the development of DJ, is what principles can be easily applied and translated to current state of the art technologies? Also, a significant amount of work would need to be done benchmarking the performance implications of the different design decisions that were made when developing DJ, since

DJ was not built for run-time performance. On top of those two questions, a plethora of future work still exists in the field of middleware database bridges, especially in the realm of emerging database technologies.

5.2 Contributions

The development of DJ resulted in three conceptual contributions to bridging a column oriented deductive database with an object oriented language. DJ's three contributions are as follows.

1. Showed that data can be represented through POJIs instead of POJOs
2. Provides a technique for mapping an object oriented language to a column oriented database
3. Implements a lightweight mechanism for declaratively writing and deploying lightweight business rules to database fields

The lessons learned while developing DJ, and the contributions are and were specifically focused on the task of bridging a deductive database with the Java programming language, however the conceptual contributions have great potential to be incorporated for both new and existing database technologies with more research.

5.2.1 Introduction of POJIs

The introduction of using interfaces instead of classes to represent data is unique for mapping an object oriented language to a database. All existing technology uses Java classes, and then persists all value changes, and queries values through point-cutting meaning that every time a method or field call occurs the middleware system must evaluate what is being asked

for and return that to the user's program. The annotated interface approach instead allows for a more declarative approach by not allowing the programmer to use fields, they must annotate a method, which logically corresponds to either retrieving a value or passing a value. Orthogonal persistence does this via getter and setter methods for private field values, but the programmer can choose to make the field public and then modify and retrieve the field value directly. This violates the object oriented principle of encapsulation. The annotated interfaces do not even provide a field declaration for the programmer, and therefore enforce object oriented coding principles when interfacing with the database

5.2.2 Object Oriented to Column Oriented Mapping

The benefits of the emerging column oriented database paradigm were presented in section 2.2. This new technology is growing in popularity as data volumes grow, and data relationships become increasingly more complex. Several companies are already in existence that make use of this technology. Unfortunately when it comes to integrating column oriented features into object oriented languages through features such as orthogonal persistence, current state of the art does not support this. The DJ system shows a methodology for representing column oriented data in the Java language while providing orthogonally persistent features. It does this potentially in two different ways. The first is through LogicBlox's conceptual model of namespaces. Since predicates or tables with a single value column can be logically grouped into a namespace, if a namespace is provided in the interface's *@Bind* annotation every column can be assumed to be in that namespace unless otherwise noted. The second way that DJ facilitates the ability to represent column oriented data in Java is by using the exact same *@Bind* annotation that is overloaded for every method in an interface. This way if the method name does not directly correspond to a column value in the database, it can be explicitly mapped by providing a different namespace, or if the database does not support the concept of namespaces, the unique column store name for that field can be provided. While the DJ *@Bind* annotation uses the "predicate" value for this the

value could easily be translated into or simply used as a "column_name" value.

5.2.3 Automatic deployment of business rules

The final key contribution of DJ is that it facilitates the automatic deployment of business rules for calculating field values in a database through it's *@Calc* annotation. Currently there is a significant industry for rule engines in the enterprise software market. This stems from the nature of relational database technology. Since the query language used (SQL) is not a complete language, the databases must be supplemented with other languages such as PL/SQL or T-SQL, and logic other than basic queries must be written as programs in these languages and stored as triggers and stored procedures on field values. Since these are procedural rather than declarative languages, the ability to just logically include rules on fields is much more complex. The benefits that DJ brings is the ability to specify a simple business rule in a declarative fashion in the source code for a field value. This keeps the burden of code and rule maintenance low. If a stored procedure is used to calculate a field value, then it must be stored and deployed outside of Java code that accesses that database. This makes it hard for developers to conceptually relate the two and maintenance is hard. Another common approach is the use of a rules engine like Drools [52]. Drools and other rule engines offer the benefits of letting a developer write their business rules in a declarative fashion while implementing the logic in a language they are familiar with such as Java. Rule engines however require querying the data from the database to operate on, and do not solve the problem of separating the logic for a simple field calculation from the Java code representation. DJ however makes it as simple and logically related as adding a formula to a spreadsheet, and handles generating all of the underlying code required. Even though relational databases do not lend themselves well to adding simple calculation rules to a field, the potential would exist with further research to apply this functionality in very basic cases. The hope is that by allowing a programmer to easily specify rules declaratively in a Java code representation of the database, that the burden of maintaining rules, enterprise code,

and a database model will be significantly easier since all three are done in the same place with the DJ system.

5.3 Future Research

Development of the DJ system opens the door for a host of future work directions. DJ was developed from the ground up, and as a result a large amount of effort went into design decisions and writing Java code. Now that a system is in place to evaluate, the potential to improve and expand it exists without the overhead of learning a completely new technology from the ground up. Some future research directions are detailed below.

5.3.1 Dynamic Queries

One key feature that DJ is lacking is the ability to generate queries at runtime, currently DJ pulls all of the information in a given predicate into memory from the database based on the data mapping. However, since the LogicBlox database exists for the purpose of working with large volumes of complex data, it may be incredibly impractical to query all of the data in a given predicate. Therefore more work is needed in the realm of allowing users to apply filters to data before a query is performed. Also, as was previously mentioned, the DJ system does not currently support the concept of joins or data operations in memory. The LogicBlox database supports the concept of temporary predicates which are the result of other queries operating across multiple predicates, and this temporary predicate only exists during the transaction. Currently DJ does not have any facilities to support extracting data from a temporary predicate created as the result of a complex query.

5.3.2 Asynchronous Queries

In data-mining and enterprise planning applications, complex calculations over large sets of data can range from minutes to possibly even days. To support such long running queries, DJ will need to provide an asynchronous client interface. This interface would enable returning control back to the caller after initiating the execution process. Upon completing execution, a callback could be triggered, notifying the client of the result. Therefore exploring how such callback interfaces should be provided and which features they should support is another future work direction that could be undertaken for DJ.

5.3.3 Advanced Business Rules

As previously noted, one of the powerful aspects of DJ's approach is that programmers are able to specify calculations (which can be thought of as lightweight business rules) in a declarative fashion at the Java level. However, the current level of support for specifying such business rules is somewhat limited. Currently DJ only supports basic arithmetic (e.g., addition, subtraction, multiplication, and division) and aggregation (min, max, total, and count) functions. However, the usefulness of the DJ system would greatly improve with support for business rules in other categories. This may require more than just engineering effort due to the fact that the ability to specify extremely complex business rules through an intuitive Java interface calls for the exploration of a large design and implementation space.

5.3.4 Transaction Management

Transaction management is a concern in all database systems. However the closer a programmer is to the database when coding, the more control they have over it. Currently the DJ middleware bridge handles all transaction management for the Java programmer, and no transaction control is provided. More work research is needed to determine if transaction management rules can be provided declaratively through the Java interfaces, or what the

best way to provide that functionality to developer would be.

5.3.5 IDE Support

Most modern software development involves the use of a rich Integrated Development Environment (IDE) such as Eclipse [35]. To aid in the ease of development for applications using DJ, exploration is needed into how providing IDE integration (e.g., an Eclipse plug-in) could facilitate the development process. Specific features that could be extremely useful include wizards for generating annotated interfaces, Intellisense support for database access, and the ability to check the correctness of deductive queries at compile time.

5.3.6 Query Batching

If ensuring high performance is at stake, handling the physical location of the database as an abstraction probably would prevent the programmer from applying manual optimization techniques to reduce query execution time. One such technique is batching multiple queries in the same network transmission. It is not clear whether determining the batch size can be done optimally in the middleware space without programmer involvement. If the programmer is involved, should they be given control over batch size? Then, what programming interface should be provided? The set of issues that need to be addressed are similar to the ones in J2EE. For example, Enterprise Java Beans (EJBs) provide both remote and local interfaces for accessing a bean. It is worth exploring whether providing different interfaces for performing local and remote deductive queries would provide performance benefits.

5.3.7 Multiple Data Sources

A deductive query might access more than one data source to obtain the answer. Consider a setup in which each store is associated with its own database, hosted on its own server.

Specifically, stores are located in New York and Seattle, while corporate headquarters are located in Atlanta. According to company policy, all software applications using individual store data are run from corporate headquarters in Atlanta. Of course, identical issues arise in relational databases, but supporting deductive queries presents an additional set of challenges. What is the optimum way to deploy business rules to the deductive engines in New York and Seattle without unnecessary (duplicate) deployments and re-configurations? How can a declarative approach support obtaining results from multiple data sources? What kind of support needs to be provided in the middleware implementation to make calculations appear as if they are being performed on the same database?

Bibliography

[1] DLinq .NET Language Integrated Query for Relational Data. Technical report, Microsoft Corporation, 2005.

[2] L. Andersen. JDBC 4.0 Specification. Technical report, Sun Microsystems Inc., 2006.

[3] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions Database Systems*, 1(2):97–137, 1976.

[4] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Readings in Object-Oriented Database Systems*, 1990.

[5] M. Atkinson, K. Chisholm, and P. Cockshott. PS-Algol: An Algol With A Persistent Heap. *SIGPLAN*, 17(7):24–31, 1982.

[6] M. Atkinson, K. Chisholm, W. Cockshott, and R. Marshall. Algorithms For A Persistent Heap. *Software Practice and Experience*, 13(3):259–271, 1983.

[7] M. Atkinson, L. Daynès, M. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4):68–75, 1996.

[8] M. Atkinson, M. Jordan, L. Daynès, and S. Spence. Design Issues for Persistent Java: A Type-Safe, Object-Oriented, Orthogonally Persistent System. *Persistent Object Systems 7 (POS-7)*, 1997.

[9] M. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *The VLDB Journal*, 4(3):319–401, 1995.

[10] C. Ballinger. Using Your OLTP Database for DSS? Technical report, Teradata, 2007.

[11] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15, 1986.

[12] C. Bauer and G. King. Java Persistence with Hibernate. 2006.

[13] J. Blakeley. Data Access for the Masses Through OLE DB. *SIGMOD Record*, 25(2):161–172, 1996.

[14] J. Blakeley. OLE DB: A Component DBMS Architecture. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 203–204, Washington, DC, USA, 1996. IEEE Computer Society.

[15] J. Bocca. Megalog: A Platform for Developing Knowledge-Base Management Systems. *Proceedings of the International Symposium on Database Systems for Advanced Applications*, pages 374–380, 1991.

[16] P. Boncz and M. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, 1999.

[17] M. Brodie. Computer Science 2.0: A New World of Data Management. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1161–1161. VLDB Endowment, 2007.

[18] CODASYL Corporation. Data Base Task Group April 71 Report. *ACM, New York*, 1971.

[19] E. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[20] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un Systeme de Communication Homme-Machine en Francais. *Research Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II*, 1973.

[21] R. Colomb. *Deductive Databases and their Applications*. Taylor and Francis Inc., 1998.

[22] Microsoft Corporation. COM: Component Object Model Technologies. http://www.microsoft.com/com/default.mspx, 2008.

[23] Microsoft Corporation. Component Database Management Systems. http://msdn.microsoft.com/en-us/library/ms718121(VS.85).aspx, 2008.

[24] Microsoft Corporation. LINQ: .NET Language-Integrated Query. http://msdn.microsoft.com/en-us/library/bb308959.aspx, 2008.

[25] Microsoft Corporation. Overview of OLE DB. http://msdn.microsoft.com/en-us/library/ms718124(VS.85).aspx, 2008.

[26] Microsoft Corporation. Rowset Programming and Object Model. http://msdn.microsoft.com/en-us/library/ms713709(VS.85).aspx, 2008.

[27] Microsoft Corporation. The LINQ Project. http://msdn.microsoft.com/en-us/netframework/aa904594.aspx, 2008.

[28] Oracle Corporation. History. http://www.oracle.com/corporate/story.html, 2008.

[29] S. Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.

[30] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88A Database Programming Language? *Proceedings of the Second International Workshop on Database Programming Languages*, pages 179–195, 1989.

[31] H. Decker. Translating Advanced Integrity Checking Technology to SQL. *Database Integrity*.

[32] M. Derr, S. Morishita, and G. Phipps. Design and Implementation of the Glue-Nail Database System. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 147–156, 1993.

[33] M. Derr, S. Morishita, and G. Phipps. The Glue-Nail Deductive database system: Design, implementation, and evaluation. *The VLDB Journal The International Journal on Very Large Data Bases*, 3(2):123–160, 1994.

[34] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2007.

[35] Eclipse Foundation. Eclipse. http://www.eclipse.org/, 2008.

[36] C. French. "One size fits all" database architectures do not work for DSS. *SIGMOD*, 24(2):449–450, 1995.

[37] H. Gallaire and J. Minker, editors. *Logic and Databases*. Plenum Press, 1978.

[38] H. Gallaire, J. Minker, and J-M Nicolas. Logic and Databases: A Response. *SIGACT News*, 18(2):52–56, 1987.

[39] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.

[40] A. Van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. *Journal of Logic Programming*, 6(1-2):109–133, 1989.

[41] A. Van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, 1991.

[42] G. Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. *Proceedings of the International Logic Programming Symposium (ILPS94)*, pages 23–42.

[43] H. Gallaire and J. Minker and J-M Nicolas, editor. *Advances in Database Theory*, volume 1. Plenum Press, 1981.

[44] H. Gallaire and J. Minker and J-M Nicolas, editor. *Advances in Database Theory*, volume 2. Plenum Press, 1984.

[45] hibernate.org. Hibernate. http://www.hibernate.org/, 2008.

[46] A. Hosking. Lightweight Support for Fine-grained Persistence on Stock Hardware. 1995.

[47] A. Hosking, E. Brown, and J. Moss. Update Logging for Persistent Programming Languages: A Comparative Performance Evaluation. *Proceedings of the International Conference on Very Large Data Bases*, pages 429–440, 1993.

[48] A. Hosking and J. Chen. Mostly-Copying Reachability-Based Orthogonal Persistence. *SIGPLAN*, 34(10):382–398, 1999.

[49] Sun Microsystems Inc. Java Persistence API. http://java.sun.com/javaee/technologies/persistence.jsp, 2008.

[50] Sun Microsystems Inc. JDBC Overview. http://java.sun.com/products/jdbc/overview.html, 2008.

[51] Sybase Inc. Sybase SQL Server. *Technical Overview. Sybase Inc*, 1989.

[52] JBoss. JBoss Drools. http://www.jboss.org/drools/, 2008.

[53] JBoss. JBoss Enterprise Middleware. http://www.jboss.com/products/index, 2008.

[54] M. Jordan. Early Experiences With Persistent Java. *Sun Microsystems Laboratories The First Ten Years 1991- 2001*, 2001.

[55] M. Jordan and M. Atkinson. Orthogonal Persistence for JavaA Mid-term Report. *Advances in Persistent Object Systems: Proceedings of the International Workshop on Persistent Object Systems (POS) and the Internation Workshop on Persistence & Java (PJAVA)*, 1999.

[56] R. Bayardo Jr, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, D. Woelk, W. Bohrer, and R. Brice. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 195–206, 1997.

[57] T. Kaehler. Virtual Memory on a Narrow Machine for an Object-Oriented Language. *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages, and Applications*, 21(11):87–106, 1986.

[58] T. Kaehler and G. Krasner. LOOM: Large Object-Oriented Memory for Smalltalk-80 Systems. *Morgan Kaufmann Series In Data Management Systems*, pages 298–307, 1989.

[59] W. Kieling, H. Schmidt, and W. Strata. DECLARE and SDS: Early Efforts to Commercialize Deductive Database Technology. *The VLDB Journal*, 3(2):211–243, 1994.

[60] R. Kowalski. Predicate Logic as a Programming Language. *Inform. Processing 74, Proc. IFIP Congr. 74, Stockholm, 569-574*, page 74, 1974.

[61] J. Kuhns. Answering Questions by Computer: A Logical Study. 1967.

[62] J. Kuhns. Logical Aspects of Question-Answering by Computer. 1969.

[63] J. Kuhns. *Interrogating a Relational Data File: Remarks on the Admissibility of Input Queries*. RAND, 1970.

[64] All App Labs. Hibernate Tutorial. http://www.allapplabs.com/hibernate/overview_of_hibernate.htm 2008.

[65] R. Levien and M. Maron. Relational Data File: A Tool For Mechanized Inference Execution and Data Retrieval. 1965.

[66] R. Levien and M. Maron. A Computer System for Inference Execution and Data Retrieval. *Communications of the ACM*, 10(11):715–721, 1967.

[67] LogicBlox. http://www.logicblox.com, 2008.

[68] R. MacNicol and B. French. Sybase IQ multiplex-designed for analytics. *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1227–1230, 2004.

[69] F. Matthes and J. Schmidt. Persistent Threads. *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 403–414, 1994.

[70] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, 2006.

[71] Microsoft Corporation. Binder Programming and Object Model. http://msdn.microsoft.com/en-us/library/ms716881(VS.85).aspx, 2008.

[72] Sun Microsystems. Sun Opens Java. http://www.sun.com/2006-1113/feature/story.jsp, 2006.

[73] Sun Microsystems. Java SE Technologies - Database. http://java.sun.com/javase/technologies/database/, 2008.

[74] H. Gallaireand J. Minker and J-M Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys (CSUR)*, 16(2):153–185, 1984.

[75] J. Minker. Logic and Databases: A 20 Year Retrospective. *Logic in Databases: International Workshop LID'96, San Miniato, Italy, Proceedings*, 1996.

[76] K. Morris, J. Naughton, Y. Saraiya, J. Ullman, and A. Gelder. YAWN! (Yet Another Window on NAIL!). *Data Engineering Bulletin*, 10(4):28–43, 1987.

[77] J. Naughton. One-Sided Recursions. *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 340–348, 1987.

[78] J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Argument Reduction by Factoring. *Theoretical Computer Science*, 146(1-2):269–310, 1995.

[79] J. O'Donahue. *Java Database Programming Bible*. Wiley Publishing Inc., New York, NY, 2002.

[80] G. Phipps, M. Derr, and K. Ross. Glue-Nail: A Deductive Database System. *ACM SIGMOD Record*, 20(2):308–317, 1991.

[81] M. Pilz. Java Timeline. http://www.ifi.unizh.ch/richter/people/pilz/java/timeline.html, 1998.

[82] R. Ramakrishnan. *Applications of Logic Databases*. Kluwer Academic Publishers Norwell, MA, USA, 1995.

[83] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2003.

[84] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL-Control, Relations and Logic. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 238–250, 1992.

[85] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL Deductive System. *The VLDB Journal*, 3(2):161–210, 1994.

[86] R. Ramakrishnan and J. Ullman. A Survey of Deductive Database Systems. *The Journal of Logic Programming*, 23(2):125–149, 1995.

[87] K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *The International Journal on Very Large Data Bases*, 3(2):107–122, 1994.

[88] R. Reiter. On Closed World Data Bases. Logic and Databases, 1978.

[89] R. Reiter. Towards a Logical Reconstruction of Relational Database Theory. *On Conceptual Modeling*, pages 191–233, 1984.

[90] K. Ross. Modular Stratification and Magic Sets for Datalog Programs With Negation. *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–171, 1990.

[91] SenSage. http://www.sensage.com, 2008.

[92] R. Simon, T. Davis, J. Eaton, and R. Goertz. *Windows 95 Multimedia and ODBC API Bible*. Waite Group Press, Corte Madera, CA, 1996.

[93] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005.

[94] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One Size Fits All?-Part 2: Benchmarking Results. *Procedings from CIDR*, 2007.

[95] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. *Proceedings. 21st International Conference on Data Engineering*, pages 2–11, 2005.

[96] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[97] A. Straw, F. Mellender, and S. Riegel. Object Management in a Persistent Smalltalk System. *Software Practice and Experience*, 19(8):719–737, 1989.

[98] Kx Systems. Kdb+ Database. http://www.kx.com/products/database.php, 2008.

[99] Kx Systems. Kx Systems - FAQ. http://www.kx.com/developers/faq.php, 2008.

[100] Symba Technologies. A History of ODBC Data Access. http://www.simba.com/simba-history.htm, 2007.

[101] D. Tsichritzis and F. Lochovsky. Hierarchical Data-Base Management: A Survey. *ACM Computing Surveys (CSUR)*, 8(1):105–123, 1976.

[102] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data Language. *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 33–41, 1986.

[103] J. Ullman. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1985.

[104] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, Inc. New York, NY, USA, 1988.

[105] J. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W.H. Freeman & Co. New York, NY, USA, 1990.

[106] J. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. WH Freeman & Co. New York, NY, USA, 1990.

[107] J. Vaghanl, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask, and J. Harland. The Aditi Deductive Database System. *The VLDB Journal*, 3(2):245–288, 1994.

[108] L. Vieille. Recursive Axioms in Deductive Databases: The Query-Subquery Approach. *Proceedings of the First International Conference on Expert Database Systems*, 1986.

[109] L. Vieille. Database-Complete Proof Procedures Based on SLD Resolution. *Proceedings of the 4th International Conference on Logic Programming*, 1987.

[110] L. Vieille, P. Bayer, V. Küchenhoff, A. Lefebvre, and R. Manthey. The EKS-V1 System. *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR92)*, pages 504–506, 1992.

[111] L. Vielle. Recursive Query Processing: Fundamental Algorithms and the DedGin System. *Halstead Artificial Intelligence Series*, pages 135–158, 1989.

[112] M. Wallace. KB2: A Knowledge Base System Embedded in Prolog. Technical report, Technical Report TR-KB-12, European Computer Industry Research Centre, Munich, Aug. 1986.

[113] H. Wang and C. Zaniolo. Nonmonotonic Reasoning in LDL++. *Kluwer International Series In Engineering And Computer Science*, pages 523–544, 2000.

[114] S. Williams and C. Kindel. The Component Object Model: A Technical Overview. *Microsoft Corporation*, 1994.

[115] C. Zaniolo. LDL++, A Second-Generation Deductive Database System. *Computational Logic*, 50:56–58, 1996.

[116] C. Zaniolo. LDL Short History. http://www.cs.ucla.edu/ldl/oldhistory.html, 1997.

[117] C. Zaniolo. A Short Overview of Deductive Databases, 1999.

[118] C. Zaniolo, N. Arni, and Q. Ong. The LDL++ system, 1992.