1

Q-DATA: USING DEDUCTIVE DATABASE TECHNOLOGY TO IMPROVE DATA QUALITY

Amit Sheth, Christopher Wood*, and Vipul Kashyap**

Distributed Information Systems Lab,
Department of Computer Science, University of Georgia,
415 GSRC, Athens, GA 30602-7404

** RRC 1H-205, Bellcore,*
444, Hoes Lane, Piscataway, NJ 08855

*** Department of Computer Science,*
Rutgers University, New Brunswick, NJ 08903

ABSTRACT

This chapter discusses an extended deductive database prototype system, Q-Data, developed by Bellcore to improve data quality through data validation and cleanup. The key technology component of Q-Data is the extended deductive database system LDL++, developed at MCC. We discuss the issues of data quality improvement, the relevance of the deductive database technology such as the LDL++ system to data quality improvement tasks, and the system architecture of the prototype. Furthermore, we describe our experiences using the deductive database technology in an on-going Q-Data trial attacking a real-world problem with test data from operational systems. Experiences related to engineering aspects of both the deductive database system and other component technologies, as well as pragmatic aspects of the implementation of Q-Data as a distributed system, are discussed.

1 INTRODUCTION

Data is considered to be a corporate resource. Good data quality is critical to efficient, high-quality operations in any enterprise. However, the issue of data

quality has received little attention in database literature [8]. A significant percentage of data in most companies are of poor quality [12]. The important dimensions of data quality include accuracy or correctness, completeness, consistency, and currentness [5].

Examples of poor data quality include errors in input data (e.g., a partial or nonexistent address), data inconsistencies (e.g., different customer billing addresses for the same customer or incorrect Zip code for the location), and unintended duplication or redundancy (e.g., multiple customer records because of different representations of the same customer such as DEC, Digital Equip. Corp., and Digital Equipment Corporation) — often contributed by duplicate or redundant data produced by different processes and organizations. Poor data quality is a result of a variety of factors, including flawed data acquisition and data creation processes, flawed data update processes, inability to enforce constraints among related data in multiple databases [7], duplicate data produced by different methods, organizations and processes, process re-engineering and company reorganizations.

Two of the most frequent manifestations of poor data quality are:

- **Inability to complete an automated process, usually in the form of transaction fall-out (errors)** due to errors, inconsistencies, and incompleteness in input data/requests and existing databases. These errors increase the cost of doing business, and require human involvement to resolve.
- **Poor customer service** due to inconsistencies and incompleteness of data. This also results in increased cost of doing business, as well as the more serious threat of losing customers.

We are addressing two aspects of managing data quality: data validation and data clean-up. *Data validation* refers to identification of data quality problems, for example by identifying inconsistent or incomplete data in inputs from users or in the existing databases. *Data cleanup* (or purification) is the process of improving data quality (usually after the data validation identifies poor quality data), for example by removing inconsistent data or making data more complete. We have built a prototype software system called *Q-Data* (where Q stands for *quality*) which uses existing deductive database technology to support data validation and clean-up. This chapter describes the Q-Data architecture, our experiences in using a deductive database technology to build a prototype system, and our experiences in applying that technology in an ongoing trial in-

volving a real-world problem that involves test data from real databases. Some of the issues discussed are related to using and testing prototype software for solving real world problems.

Not all aspects of data quality can be determined or enforced by a computerized system. However, deductive database technology can be a natural choice for addressing many data quality issues, because deductive databases provide a natural way of:

- capturing business rules, practices and constraints that define data validation and cleanup rules, and
- integrating those rules with access to databases where a significant portion of corporate data reside.

Some of the solutions to data quality problems offered by deductive database technology may be partial. While some clean-up operations can be performed automatically in a batch mode of operation, others may involve interactive human participation. Thus, besides rule processing and data access, Q-Data also has a significant user interface component.

We selected the LDL++ system from the Carnot project at the Microelectronics and Computer Technology Corporation (MCC) for the deductive database component of Q-Data. In addition to the functionality of the Logic Database Language (LDL) [9], the LDL++ system also supports definition of LDL predicates implemented as C++ functions. (A predecessor to the LDL++ system is described in [4].) This extension to basic deductive database technology is important for certain processing-intensive operations not supported directly by LDL. We also extended the LDL++ system to interface with Prolog. Thus the LDL++ system provides us with a rich processing environment that supports both declarative logic based computation in its two flavors viz., top-down (Prolog) and bottom-up (LDL), and procedural data manipulation using C++. Our implementation includes interfaces to Ingres^{TM} [1] and Oracle^{TM} database management systems (DBMSs) using Extensible Services Switches (ESSs) developed in the Carnot project at MCC. An additional interface to DB2^{TM} is being developed, though not yet used in a trial.

[1] Ingres is a trademark of Ask, Inc. Oracle is a registered trademark of Oracle Corp. DB/2 is a trademark of International Business Machines Corp. Sybase is a trademark of Sybase Corp. Any omissions were inadvertant.

In a significant trial involving the prototype Q-Data system, we developed a set of rules to identify incompleteness and inconsistencies in data from a customer database of a client of Bellcore, display such data to a user, suggest correct replacement values, and update the database with the corrected information. Rules can be written that identify dirty data (i.e., data that violate integrity constraints) and used to identify inconsistencies during data validation, or they can be written to describe the integrity constraints, in which case the system searches for exceptions to such constraints. Enforcing data consistency and completeness rules, such as the ones developed for this trial, against an existing database is an excellent real-world test of deductive database concepts and software.

A goal of our work has been to determine whether deductive database systems are suitable for data validation and cleanup applications, and to evaluate the performance of the LDL++ system, to determine whether it is ready to solve a class of real-world problems related to data quality. Some of the needed improvements go beyond the scope of the basic deductive databases technology (viz., issues of distributed and client-server systems). As discussed in some detail in this chapter, we found that the technology holds significant promise, although the current implementation needs some engineering improvement. Deductive database technology has been advertised to be of great value for complex, data intensive applications; we believe ours is one of the early efforts to substantiate its utility for a class of real world applications.

The rest of this chapter is organized as follows. Section 2 briefly discusses some of the related work. Section 3 describes the overall data cleanup problem faced by a typical customer, and identifies several specific data cleanup tasks that are used as examples. Section 4 describes the software architecture for the Q-Data prototype system. Our experiences analyzing available technologies as components of Q-Data, and designing, building, and testing the prototype are discussed. We also describe in greater depth the strengths and weaknesses of LDL, Prolog, C++ and SQL languages and their implementations, for searching and control, database access, and rule specifications. In Section 5, we consider pragmatic and engineering issues. We present our conclusions in Section 6.

2 RELATED WORK

Our investigation in data validation and cleanup is driven by specific needs of our clients. These needs are similar to those discussed (albeit at a high-level)

in [10] and [16]. [10] discusses developing a (new) corporate subject database consisting of customer information by cleaning up data from multiple legacy systems. One application discussed in [16] involves verifying that related data in three legacy application systems are consistent. Another application discussed in [16] involves establishing correct location information in the input data with the help of a reference location database, and using that reference database to clean previously created location data in an application database.

There is an extensive literature on deductive databases, ranging from loose-couplings of Prolog interpreters to relational DBMSs, through interfacing parallel logic languages (e.g., Parlog) to DBMSs [2], and to compiled, set-at-a-time processing of logic queries against a database. Some of the significant efforts that also involve significant prototyping and in some cases performance evaluation include BERMUDA [6], BrAID [17], the LDL++ system [1] and Coral [14, 18, 15]. Our choice of the LDL++ system was guided by the merits of the system in functional terms, especially the ability of the LDL++ system to access commercial DBMSs of interest to us and its interface to C++, as well as very important pragmatic considerations, such as:

- availability of technical support,
- experience with and use of the LDL++ system in other (although perhaps less demanding) applications,
- interests of our clients in the LDL++ system, and
- availability of the source-code for possible future productization or long-term maintenance.

3 EXAMPLES AND A TRIAL APPLICATION OF DATA VALIDATION AND CLEANUP

In this section, we present an overview of our client's application. We then discuss classes of data validation constraints, which, when violated, indicate that data has become corrupt, or "dirty". These constraints are illustrated with examples taken from the application. LDL rules for identifying "dirty" data, i.e., when a piece of data violates the validation constraints are also demonstrated. The data cleanup rules, i.e., the rules which specify the "fix" for "dirty" data are also discussed.

3.1 An Example Application

In this application, "dirty" data from existing operational systems is "cleaned" and a new shared corporate database is populated with "clean" data. The data relates to the client's customers, and the telephone services that the client provides to its customers at different locations. The data is in several tables in a relational DBMS (some of these tables are populated by extracting data from non-relational databases of legacy applications). Figure 1 shows the most important fields for five tables in this database. The key columns for each table are in boldface. For our trial of Q-Data, our client provided a significant sample (13 Megabytes) of the database taken from operational systems.

The algorithm used by the client to add information to the database sometimes creates new locations and enterprises when it should be adding to existing locations and enterprises. Frequently, this is caused by names being entered with spelling errors, abbreviations, or different word boundaries. The client supplied a "soft-match" algorithm that could match two strings in spite of spelling errors, abbreviations, and different word boundaries. One of the main requirements is to search the database and remove the duplicated entries. In addition, the client wanted a flexible system, so that they could implement additional validation and cleanup rules as they were established. The rules could also be used to clean up different databases.

3.2 Examples of Data Validation Constraints

We identified three broad classes of data validation constraints. The classes are domain value constraints, quantitative constraints (which include, as special cases, uniqueness constraints and referential integrity constraints), and miscellaneous other constraints. Additional data validation constraints could be generated by way of good data modeling techniques.

Domain Value Constraints

This simple type of constraint limits the set of values that can appear in a given column of a given table. An example domain value constraint is that a column contains an integer between 0 and 100; another is that it contain either the letter 'Y' or the letter 'N'. Most modern DBMSs will enforce value domain constraints as tuples are added or modified. However, we frequently need to validate and clean data that were generated by legacy applications that were built using DBMSs that provided little support for enforcing semantic integrity constraints.

Example:

Each value in the column 'Type' in the 'Addresses' table should be one of 'BLG', 'LST', or 'SER'. This would be expressed in LDL as:

```
AddressTypeViolation(TelNumber, Adrs_type) <-
        addresses(TelNumber, _, Adrs_type),
        Adrs_type ~= 'BLG',
        Adrs_type ~= 'LST',
        Adrs_type ~= 'SER'.
```

Quantitative Constraints

A quantitative constraint specifies that the number of items that satisfy a given condition fall within a given range. Uniqueness constraints and referential integrity constraints are two special cases of quantitative constraints, where the condition is existence, and the quantity is either one (for uniqueness constraints), or one or more (for referential integrity constraints).

Examples:

- There should be at least one billing name, and exactly one billing address for each location. The constraint on addresses cannot be expressed as a referential integrity constraint, because the association from LocCode to billing name and billing address is through one of the TelNumbers. The LDL code for the billing name constraint is:

```
billingNameViolation(Location) <-
        billNamesPerLocation(Location, BillNameSet),
        cardinality(N, BillNameSet),
        N > 1.

billNamesPerLocation(LocCode, <BillName>) <-
        locations(_, LocCode),
        numbers(TelNumber, LocCode),
        names(BillName, 'BLG', _, TelNumber).
```

- Each location should only appear once.
- Each enterprise should be represented by only one row in the 'Enterprises' table. This requires the use of soft match on the EntName column.

Uniqueness constraints

Uniqueness constraints, where the value of attributes in each row of a given table must be unique, are enforced by most DBMSs. Defining a column as a key for a table implies that all the values in that column should be unique. Examples of uniqueness constraints for this application include:

- 'EntCode' in the 'Enterprises' table, and
- the set of attributes (TelNumber, Type, Number) in the 'Names' table.

Referential Integrity Constraints

Referential integrity constraints specify that values of an attribute in one table must occur in another table (possibly being managed by a different DBMS).

Examples:

- Every row in the 'Locations', 'Numbers', 'Names', and 'Addresses' tables should be represented by a row in the table to the left ('Enterprises', 'Locations', and 'Numbers'). The first two are expressed in LDL by the following:

```
LocationRefIntegViolation(EntCode)<-
        Location(EntCode, _),
        ~Enterprise(_, EntCode, _).

NumberRefIntegViolation(LocCode) <-
        Number(_, LocCode),
        ~Location(_, LocCode).
```

- Every row of 'Enterprises' and 'Locations' must have at least one row referring to it in the table to its right ('Locations' and 'Numbers').
- There should be at least one listing name, and one listing address for each Telephone Number.

Other Constraints

Of course, specific problems may involve constraints that do not fit into any ofthe above categories. These are general constraints.

Examples:

- Rules about formatting across columns (i.e., that the street number and street name components of an address should be in separate columns).
- Value of one attribute should indicate which row in a set of rows has the largest (or smallest) value of another attribute.

3.3 Examples of Data Cleanup Rules

The second and third quantitative constraints (in Section 3.2.2) that require locations only have one entry in the 'Locations' table, and to have only one entry in the 'Enterprise' table for every customer (enterprise) is a significant

Figure 2 Merging Enterprise Codes

challenge facing our client. Figure 2 shows an example from the Enterprise, Location, and Telephone Number hierarchy, where an enterprise is represented by two different enterprise codes that should be merged. Once they are merged, the corresponding Location codes must be examined to identify location information that must be merged.

The following strategy is used for dealing with potential duplication of enterprise information:

- When two tuples from the 'Enterprise' table (i.e., two distinct Enterprise Codes) have similar names, neither name appears on a reference list of franchises, and at least one address is associated with both Enterprise Codes, the enterprises really are the same, and should be merged together.
- Two tuples from the 'Enterprise' table have names that are similar, neither name can be found on a reference list of franchises, and there are no addresses common to the two. In this case, it is likely, but not definite, that the enterprises are the same. A user should be presented with the information, and asked to verify that the similar names really do refer to the same enterprise.

The requirement that all addresses should be valid warrants additional explanation. The customer has a reference database of geographical information, which includes the names of all the streets within villages, townships, boroughs, and cities. A different DBMS supports this database.

3.4 Maintenance of Clean Data

While planning a project to clean dirty data, a system architect should also consider the causes of data becoming dirty, and take preventive action to see that this does not happen again. Once the databases are clean (i.e., all relevant constraints are satisfied), the application programs and DBMSs must maintain that state of cleanliness.

One approach to this would be to create a data validation module that operates on input requests. This module would examine data contained on an update request, evaluate the data validation rules, and consult the current contents of the database to determine whether the request can be satisfied, or must be rejected. This approach does not address all possible scenarios (e.g. data becoming out-of-date with the passage of time).

4 SYSTEM ARCHITECTURE AND FEATURES

This section describes the functional architecture and the features of the Q-Data prototype. The partitioning of the functionality of the system into three main layers is discussed, and the functionality of each layer is identified. The choice of an appropriate technology and language for the rule specification and processing layer is discussed. Alternatives for external database access are discussed, and the component choices for the various pieces of Q-Data are identified.

4.1 Overview of the Architecture

We derived the requirements for this project after discussions with the customer about their needs. Based on these requirements, we decided that the implementation of the prototype would have to tackle three sets of issues: user interface, rule/program specification and processing, and database access. The Q-Data architecture thus follows the industry practice of partitioning software functionality into user interface, information processing, and data storage and management as suggested in the OSCA^{TM} Architecture[3].

As we started this project, we noted that our requirements for access to data, and searching and manipulating that data, could be met, at least function-

ally, by using the software developed by the Carnot project and its predecessor project on deductive databases at MCC. Specifically the LDL++ system combines a compiler for a declarative, logic-based programming language (LDL) with the ability to access functions written in C++, and the means to access external databases.

Currently, the LDL++ system can access data in several commercial relational DBMSs and an object-oriented DBMS. Access to data in relational databases is important since it is easy to access and manipulate data with the industry standard query language SQL. Also, some RDBMS vendors provide gateways that enable (indirect) SQL access to additional data on a mainframe resident DB2 DBMS. Data for our trial application were in a relational database. A significant portion of telecommunications data also reside in databases under the control of application systems called Operation Support Systems (OSSs). In future, by creating access functions in C++ and using other interface software developed at Bellcore, we can enable the Q-Data system to access mainframe based OSS data from workstation based environments.

4.2 System Architecture

In this section, we describe the features of each of the three layers identified in the previous section: user interface, rule/program specification and processing, and database access. Figure 3 shows the Q-Data system architecture. Items in dotted lines have not yet been implemented.

Graphical User Interface

The Q-Data architecture features a Graphical User Interface that:

- enables the user to input information that the processing layer will check against the databases, using the data validation and cleanup rules,
- shows the user available validation (data analysis) and/or data cleaning choices (queries),
- allows the user to invoke the validation and/or cleanup operations,
- presents data whose quality is suspected to be poor to the user for review, and

Figure 3 Q-Data System Architecture

- provides a mechanism for the user to make corrections with the assistance of the system (assisted cleanup) or without (manual cleanup). Some cleanup can be totally automatic without user involvement. Section 4.5 describes further these modes of data validation and cleanup.

Rules/Program Specification and Processing Layer

Application experts or database administrators create rules and programs that capture business-specific rules or general (business-independent) constraints regarding data quality. The processing component then searches the data sources to find all data that satisfy (or, depending on the semantics, violate) the rules. Optionally, automatic cleanup can also be performed using appropriate rules and programs. The key features of this layer are:

- Rules are written in powerful, logic-based languages (Logical Data Language (LDL) and Prolog) and may involve predicates implemented by functions written in a procedural languages (C and C++), to search for and/or manipulate data stored in DBMSs.
- Rules can be dynamically added in some cases.
- Any rule can access (read/update) data in multiple databases if the LDL++ system provides access to the involved databases.
- Invocation of each query is treated as a separate transaction, so that all side-effects are committed or rolled back together.

The LDL++ system/C++ External Predicate Interface

The LDL++ system enables the user to define and implement predicates using the C++ language. The system provides interface functions that enable external predicates to:

- examine and create data items that cross the interface between the external predicate and the LDL++ system,
- discover whether the predicate is being called for the first time, or the LDL++ system is backtracking into the predicate because a later predicate failed,
- maintain a private state between predicate invocations,

- invoke other predicates (these could be implemented within the LDL++ system or as external predicates), and
- be dynamically linked into a running process of the LDL++ system.

In our trial, we used the external C++ interface to interface the LDL++ system with a stand-alone Prolog implementation of the soft match algorithm.

Database Access Layer

The database access layer provides access to data in source, reference, and target databases. The features of the database access layer include:

- Database specification separate from rules (i.e., rules are written without regard to which database contains the data).
- Client/Server or distributed access to several relational DBMSs, either through an interface built into the LDL++ system (for Sybase^{TM}, and possibly DB/2), or through an Extensible Services Switch (ESS), which enables the LDL++ system to communicate with multiple DBMSs from different vendors through a single interface.
- Preferred database access is through Remote Data Access (RDA) standard via SQL Access Group (SAG)/Call Level Interface (CLI) Application Program Interface (API).
- Use of industry standard communication technology where possible (TCP/IP).

We used the separation of the database specification from the rules during development, and the Client/Server access to an Oracle DBMS through the ESS and Oracle's SQL*NET product.

4.3 Alternatives for the Rule/Program Specification and Processing Layer

We now enumerate and discuss the various language and technology choices for the various data validation and cleanup tasks to be handled by the rule specification and processing layer. We have identified two critical functions for data cleanup tasks for which language and technology choices have to be made:

- The primary function of all data validation and cleanup tasks is query processing that involves retrieving and updating all the data which satisfies the conditions specified in the data validation rules, followed by updating the database so that all of the integrity constraints on the database are satisfied. We discuss the advantages and disadvantages of using Prolog, LDL and Embedded SQL for this function.
- The other critical function is the comparison of different pieces of data in the system. This typically is expressed in the form of comparisons between and operations on strings. We discuss the advantages and disadvantages of using Prolog, LDL and C++ for this function.

Prolog

In this section we investigate the advantages and disadvantages of using Prolog for the two critical data cleanup functions: query processing and string comparison.

Prolog for query processing: Disadvantages

One major sub-task in Q-Data was retrieving all data which satisfied the conditions specified in a rule to retrieve the "dirty" data. We found Prolog to be unsuitable for the following reasons:

- Prolog semantics retrieve the first result of a query; the programmer must perform extra work [2] to retrieve all possible solutions to a query.
- The programmer must explicitly understand and account for Prolog's rule selection mechanism while writing Prolog rules, and use procedural constructs like "cut" to direct the Prolog system toward the desired goal. This detracts from the advantages of declarative programming, and limits the ability of the Prolog interpreter to optimize query processing.
- Prolog does not guarentee that any particular query will terminate.
- Prolog does not guarantee that any particular query will return all matching data items.

[2] A simple generate and fail loop around a query may not be sufficient, because the rule produces valid tuples, but still fails. Because of the failure, predicates that use such constructs must never be used to compose more complex predicates, and must be segregated from the main logic of the program.

Prolog for string matching: Advantages

Another major sub-task in Q-Data was to determine whether two strings were duplicates of each other. A special kind of rule called a "soft string match" was needed which would match two strings even if there were misspellings, abbreviations, letter transpositions and/or different word order in the two strings. We found Prolog to be very suitable for this sub-task because:

- The logic program for the "soft string match" naturally lends itself to a top down computation. It can be characterized by the classical "list append" problem. Prolog having a top-down model of computation is suitable for the purpose.
- Prolog would evaluate only one computation path in the proof tree and would stop as soon as it evaluates the first possible way. A bottom-up language like LDL would evaluate all the computation paths of the proof tree, i.e., it would evaluate all possible ways two strings might "soft match" with each other.
- List manipulation predicates (append, cons, etc.) in Prolog are implemented using C functions and hence are more efficient than equivalent processing in the LDL++ system.

We were provided with a Prolog implementation of the soft match algorithm by our client. We ported it to LDL and discovered that the Prolog version runs two orders of magnitude faster than the LDL version. During the development of the Q-Data system, we developed C++ code that enables the LDL++ system to pass data to a rule implementation written in Prolog. The effort was entirely successful, and the overhead to access the Prolog implementation is low.

C++ (A procedural language) and SQL

In this section we investigate the advantages of using C++ for the important data cleanup function of string matching. We also investigate the disadvantages of using SQL embedded in C++ for the data cleanup function of query processing.

Embedded SQL with a procedural language (C++) for query processing: Disadvantages

Another option we considered was to embed the data retrieval statements (SQL

in this case) within a procedural programming language (e.g., C++). Disadvantages of this approach are well known:

- C++ is not a declarative language. The semantics of a query are never explicitly stated; they only emerge from the process of retrieving data. We are interested in building a prototype in which the user has the freedom to modify old data validation and cleanup rules and specify new data cleanup rules with ease. The user would have to delve in C++ code that directs the search procedure to change the data cleanup rules.
- In C++, the ability to specify a program independent of the database is absent. In addition, the programmer has the additional burden of declaring temporary variables, storing intermediate results, and managing SQL cursors. For example, user would have to write a new program every time a new database is added to the system.

C++ for operations on Strings: Advantages

The C and C++ libraries for manipulating strings are well known, frequently used, efficiently implemented, and tested by time. They are clearly the tool of choice for manipulating strings of characters. The external predicate interface in the LDL++ system enables developers to use these libraries for string manipulation in LDL programs.

The LDL++ System

In this section we investigate the advantages and the ease of specifying the rules in the LDL++ system, which has a declarative framework.

The LDL++ system for query processing: Advantages

The LDL++ system provides set-at-a-time semantics which retrieves all solutions of a query at a time, and implements a powerful, purely deductive language (LDL) for expressing rules. In addition, LDL takes into consideration the issues of updating databases (and other side-effects), and supports these inherently procedural features within the overall declarative framework. While other languages used in deductive databases can also support these processing requirements, using the LDL++ system offers two distinct advantages: integration with C++, and access to commercial DBMSs.

An important advantage of using LDL over embedded SQL is the ease of programming, which is a consequence of the declarative nature of LDL. The additional burden of declaring temporary variables, storing intermediate results, and managing SQL cursors is avoided. C/C++, SQL, or SQL embedded in C/C++ cannot easily specify some of the complex data manipulation involved in data quality rules. SQL code, even when it is equivalent to several rules of LDL, could run a few pages and may require many levels of nesting.

4.4 Alternatives for the External Database Access Layer

Data to be cleaned resides in a variety of commercially supported relational database management systems. The query processing system must efficiently access data in a variety of DBMSs. We now discuss the advantages and disadvantages of systems built around Prolog, SQL embedded in a procedural language, and LDL.

Prolog

- There is an impedance mismatch between Prolog and databases being searched by the rule processing layer. Prolog retrieves and processes facts a tuple at a time (e.g., see [11]) while database queries in SQL return answers a set at a time. This constrains designers who interface Prolog systems to DBMSs, to query those databases a tuple at a time. This increases the load on the DBMS and the network, or requires optimized data retrieval that combines the evaluation of several predicates into a single retrieval from the database.

SQL Access Group Call Level Interface (SAG/CLI) and Embedded SQL

- There is an impedance mismatch between SQL (set-at-a-time) and C++ (tuple-at-a-time) which requires additional programming effort and specialized buffering strategies.
- Applications using SAG/CLI or Embedded SQL will experience great difficulty interfacing with DBMSs from multiple vendors. Each vendor's SAG/CLI library is required to use the same name for the standard functions to access the vendor's DBMS. This means that it is impossible for an

application to simultaneously access databases from multiple vendors using the SAG/CLI. Also, there is no assurance that different vendors' methods and tools for using embedded SQL will use methods that can coexist.

LDL

- Databases return all solutions set-at-a-time. LDL follows the set-at-a-time semantics. Typical database applications lend themselves naturally to bottom up computations and can be characterized by the "fibonacci number generator" programs. LDL semantics are defined as a bottom-up evaluation and hence is suited for database applications.

4.5 Component Choices

The LDL++ system is a good choice for the query processing and database access modules, because it allows us the flexibility to match languages that offer different semantic features to different aspects of data cleanup problems.

LDL is the best overall choice for writing rules for general pattern matching, and filtering the greatest amount of data, because the LDL++ system's query optimization can combine several rules into a single SQL statement, which more tightly specifies the data needed from the DBMS, and lets the DBMS perform the actual selection of the data. This reduces the amount of data that the DBMS returns to the query processing module, and enhances overall performance since the DBMSs are highly optimized for selecting data.

C++ is appropriate for rules that are naturally expressed procedurally, or when the rule can be based on existing C++ code (such as the string library).

Prolog is appropriate for rules where several of the following conditions are met:

- the rules are best expressed declaratively,
- there is a significant amount of computation needed to evaluate the rules,
- only one or a few solutions are returned, and
- there is minimal need for access to stored data.

5 PRAGMATIC AND ENGINEERING EXPERIENCES

In this section, we discuss our experiences in developing the Q-Data system that reflect on practical engineering issues. First, we discuss three different modes of data cleanup operations: manual cleanup, assisted cleanup, and automatic cleanup, and discuss the advantage of batch validation with interactive cleanup. Then, we describe the physical system configuration. Next, we describe the impact of data location on system performance, and an important "quick fix" that enabled the LDL++ system to use SQL aggregate operators to significantly improve the performance of an important type of query. We then discuss techniques for rewriting LDL queries to improve performance, and describe where we believe the LDL++ system could perform this tuning automatically. We conclude with a summary of the key engineering results.

5.1 Modes of Data Validation and Cleanup Operations

We have identified three approaches to cleaning invalid data: manual cleanup, assisted cleanup, and automatic cleanup. Manual cleanup is the simplest approach, as it only involves presenting the user with the invalid data, and places responsibility for correcting the data on the user. In assisted cleanup, the validation and cleanup system proposes replacements and additions to the databases that will make it correct (consistent and complete). The user verifies the proposed changes or selects from among several choices proposed by the system. Automatic cleanup means that the system modifies the database to make it correct (leaving an audit trail) without interacting with a user. Assisted and automatic cleanup systems are possible when the rule developers can anticipate the nature of the dirty data, and write rules, in advance, that can (partially or completely) correct the problems.

Batch Validation with Interactive Cleanup

Many data validation tasks that we prototyped take significant amounts of time to process, often from 30 seconds upwards, at least given the current implementation and hardware. Users find it unacceptable to wait this long for a system to return with some results, even though manually performing the same process would have taken two or three orders of magnitude longer. Therefore, it is important, in a real system, for the user to be able to invoke multiple

Figure 4 Buffering Dirty Data

data validation queries in a batch mode, and then go on to other tasks while the system searches for invalid data. Since the user may be working on other tasks, results of data validation queries must be stored in a buffer or a file. The user can then peruse these results and perform cleanup at any time after the results have been obtained. Figure 4 shows the process for performing batch validation, buffering the data, and then interactively cleaning the dirty data in manual or assisted modes.

5.2 Physical System Configuration

The Q-Data prototype is physically implemented on multiple computers, and runs on a busy network; the Oracle DBMS is serving many databases and many users. Figure 5 shows the current distribution of functionality to different computers. The Oracle and Ingres DBMSs are fixed on their respective hosts by licensing agreements. The Ingres ESS must be co-located with the Ingres DBMS, because we do not have the Ingres networking software. Because the Oracle DBMS is shared among many users and applications, the system performs better when the Oracle ESS is on a separate computer. This was determined empirically, and indicates that retrieving data from the database, and

Figure 5 Physical Implementation of Q-Data Prototype

processing it through the ESS[3] is a CPU bound task, and applying different CPUs to the task helps performance more than the communication overhead hurts it. We are also assuming that the Graphical User Interface will run on a user's workstation, while the LDL++ system and Prolog processes will be running on a faster server. We are investigating whether co-locating the LDL++ system and the Oracle ESS would improve performance.

5.3 LDL++ System Performance

An important consideration is that it is not sufficient to write queries that declaratively specify the desired results; the queries must run in a reasonable amount of time, consuming a reasonable amount of system resources (such as memory, network bandwidth, or processor cycles).

LDL evaluates rules by using matching (rather than unification) with magic set optimizations to propagate constants into the queries. When a failure is encountered, the control will backtrack through rules that have been matched, to attempt to make further progress by rebinding variables to other values.

[3]Current behavior and knowledge of the components suggest to us that the ESS implementation consumes processor cycles and time that greatly exceed the functionality we need for data access. We are investigating lighter-weight alternatives. However, throughput also increases when a comparatively light client process, Oracle's ad-hoc query tool SQLPLUS, is moved from the host with the Oracle DBMS to a user's workstation.

The time the LDL++ system requires to complete a query has two components: the time required to access the relevant data, and the time required to process the data within the LDL++ system. The data access time can be divided into two components: the time required for a DBMS to process the SQL query, and the time required to communicate the results back to the LDL++ system. For data validation and cleanup applications, the data access time usually dominates. For simple rules (such as referential integrity validation) on a well-configured database, the data communication time dominates the DBMS processing time. On other queries, where the DBMS references more tables and returns fewer tuples (i.e. the selectivity of the query is higher), the DBMS processing time dominates.

Performance of the current system

This section briefly describes the current level of performance of the LDL++ system. Consider the following query that identifies rows that violate referential integrity:

```
export refViol(KeyA, RefConstraint).
refViol(KeyA, RefConstraint) <-
        table(KeyA, _, RefConstraint, _),
        ~refColumn(RefConstraint).
```

If table() and refColumn() are both tables of ground facts exported by the same DBMS, then this query will be collapsed into a single SQL statement that can be efficiently processed by the DBMS. The only tuples communicated will be those which satisfy the overall query; data communication time is optimal. On the other hand, if the two tables are supported by different DBMSs, then the LDL++ system has to perform the join. If the database systems can be modified so that a temporary copy of the refColumn relation is loaded into the same DBMS as the table being checked ahead of time, both of these costs can be almost entirely avoided. The join operation gets pushed into one DBMS query, and the only tuples returned by the database would be those that satisfy the overall rule.

The LDL++ system performs joins across DBMSs by projecting out needed columns of the first table, and then, for each row in table, attempting to select a matching row from the second table. For the example above, the LDL++ system projects out the first and third columns of table(); then, for each row,

the LDL++ system selects a matching row from the table refColumn(). If this selection fails, the pair (KeyA, RefConstraint) is added to the solution.

This approach to cross-DBMS joins generates a tuple that is communicated back to the LDL++ system and a query against the second DBMS for each row in the first table. Both of these times are significant in the current implementation. The DBMS/ESS/LDL++ combination takes an average of 450 milliseconds to process each SQL query, and approximately 40 msec. to retrieve each 136-byte tuple for a sustained read from a table (e.g. "select * from tableName"). Note that these measurements are made with a DBMS server being used by multiple clients, and accessed over a busy network. In comparison, in the same environment, the vendor's ad-hoc SQL query tool takes approx. 10 msec. per tuple.

Modifications to LDL and the LDL++ system for improved performance

At the start of this project, the LDL++ system did not take advantage of the aggregate operators in SQL (count, sum, min, max, and avg). Because of the communication and query overhead described above, the LDL++ system took orders of magnitudes longer to process quantitative validation queries than the DBMS took to process an equivalent SQL query that uses the aggregate operators. To resolve this shortcoming, MCC extended LDL to enable the use of any SQL aggregate operator in the same way that set grouping is performed. Currently, these operations are only implemented through SQL (and hence, only operate on ground facts stored in an SQL DBMS). An implementation for the internal database may be developed in the future (e.g. see [17]).

An example of the LDL constructs that take advantage of SQL aggregate operators is in Section 5.4, "Eliminating unneeded evaluations".

This solution, while sufficient for the short run, is unattractive in the long run. This is a special-purpose patch to the LDL language to take advantage of particular data access features. By requiring a separate rule to perform an aggregate operation, the task of developing LDL rules is made more complex. In addition, there are now two different language constructs for performing an aggregation operation; one construct can only be used for facts stored in an external database, while the other construct will only perform acceptably on facts or relations stored within the LDL++ system. This is an unnecessary complication of the language. Ideally, the LDL++ query compiler should look

ahead to determine if and where values are used, and use this information to determine whether a partial evaluation of an expression is in order (for example, only recording the number of elements in a set, rather than the value of each member of the set.)

5.4 Performance Tuning and Future Performance Improvements

The LDL++ system promises declarative programming — that the programmer need only describe the desired solution, and not be concerned with the procedures used to arrive at the solution. However, the way a query is expressed in the LDL++ system has significant impact on the time required to process the query. This section describes three techniques for rewriting LDL queries that have been found useful in tuning the performance of the LDL++ system:

- Minimizing communication by reordering and rewriting queries,
- Eliminating redundant subexpressions, and
- Eliminating unneeded evaluations.

To tune the performance of a query, the programmer must be aware of the process that the LDL++ system uses to evaluate queries, and use this knowledge to rewrite queries to reduce their overall cost.

In the future, the LDL++ system could be improved to perform automatically the suggested analysis during the compilation stage, to improve query performance without forcing an extra burden on the programmer. Techniques that have been well-developed in the compiler and query optimization communities need to be applied to deductive databases as well.

Minimize communication by reordering and rewriting queries

The data being retrieved from a database must be carefully analyzed to prevent fetching the same tuple twice. Use the rules of logic to rewrite the query so that each query sent to the DBMS selects a disjoint subset of the needed data. The following example uses the schema presented in Figure 1.

A TelNumber has a valid listing address if:

- the Addresses table has a listing name and a listing address for the TelNumber, or
- the Names table for the TelNumber starts with the string "NON PUB".

The query emptyListAddr returns all the TelNumbers where there is no valid listing address. The original LDL code was:

```
% This implementation is inefficient

emptyListAddr(TelNumber) <-
        numbers(TelNumber, _),
        ~okListAddr(TelNumber).

okListAddr(TelNumber) <-
        names(ListedName, 'LST', 1, TelNumber),
        stringCompare(ListedName, 'NON PUB', 7).

okListAddr(TelNumber) <-
        names(_, 'LST', 1, TelNumber),
        addresses(TelNumber, _, 'LST').
```

Because okListAddr() is built using the external stringCompare() predicate, the LDL++ system cannot compress this into a single SQL statement[4]. The LDL++ system evaluates this version of emptyListAddr() by copying the entire TelNumber() column from the table numbers, and then launching two queries (one looking for an address, the other looking for a name) for each row. This is very inefficient. An equivalent query runs much more efficiently:

```
% This version is efficient

emptyListAddr(TelNumber) <-
        numbers(TelNumber, _),
        ~addresses(TelNumber, _, 'LST'),
```

[4]Support for SQL string comparison using "LIKE" and wild cards may be added to LDL and the LDL++ system in the future.

```
        ~names(_, 'LST', 1, TelNumber).

emptyListAddr(TelNumber) <-
        numbers(TelNumber, _),
        names(LN1, 'LST', 1, TelNumber),
        ~addresses(TelNumber, _, 'LST'),
        ~stringCompare(LN1, 'NON PUB', 7).
```

This form of the query minimizes the number of tuples that are fetched from the database. Every tuple that satisfies the first rule above satisfies the overall query, and must be fetched from the database. No tuple that satisfies the first rule satisfies the second rule, so no tuples are fetched twice. This also assures that the set of tuples that is passed to stringCompare is minimal. A total of two SQL queries are sent to the database.

Eliminating redundant sub-expressions

Performance of the LDL++ system query can frequently be enhanced by eliminating redundant sub-expressions. Care must be taken, though, to assure that eliminating common sub-expression does not increase the total communication burden. Note that the change shown in the previous example improves performance, but also calls for the evaluation of numbers() twice. In that example, the LDL++ system combines, into a single query, the evaluation of the rule numbers() with the subsequent statement or two. So the cost of the extra evaluation is minimal.

When intermediate results cannot be pushed down into the database, they should be stored, rather than regenerated, as in:

```
rule(A) <-
        subrule1(A, B),
        subrule2(A, C),
        ...

subrule1(A, B) <-
        generate_intermediate(A, Z),
        use_intermediate1(Z, B).

subrule2(A, C) <-
        generate_intermediate(A, Z),
```

```
        use_intermediate2(Z, C).
```

This evaluates generate_intermediate() twice. To avoid this, the query should be rewritten as:

```
rule(A) <-
        generate_intermediate(A, Z),
        use_intermediate1(Z, B),
        use_intermediate2(Z, C),
        ...
```

Common sub-expressions may appear where an if-then-else construct would be more efficient. For example:

```
rule(A) <-
        test(A),
        success_rule(A).

rule(A) <-
        ~test(A),
        failure_rule(A).
```

evaluates test(A) twice. LDL features an if-then-else structure that can be used to eliminate this extra evaluation. The query should be rewritten as:

```
rule(A) <-
        if(test(A)
                then success_rule(A)
                else failure_rule(A)).
```

Eliminating unneeded evaluations

A programmer can eliminate evaluations of expressions by looking ahead to determine whether the full result of a given function is actually needed. Writing code that makes use of the aggregate functions of SQL is an example of this. Consider the following code:

```
multipleAddrs(LocCode, N)<-
        numbers(LocCode, _),
        addrs(N, 'BLG', LocCode),
        N > 1.

addrs(N, Type, LocCode) <-
        fullAddrSet(AddrSet, Type, LocCode),
        cardinality(AddrSet, N).

fullAddrSet(<Address>, Type, LocCode) <-
        numbers(TelNumber, LocCode),
        addresses(TelNumber, Type, Address).
```

The LDL++ system translates this into the following sequence of SQL statements:

```
DECLARE LdlEssQueryCursor0 CURSOR FOR
SELECT DISTINCT ldl_t0.locCode FROM locations ldl_t0

OPEN LdlEssQueryCursor0
FETCH LdlEssQueryCursor0
<<2223525024894

DECLARE LdlEssQueryCursor1 CURSOR FOR
SELECT DISTINCT ldl_t1.address
FROM numbers ldl_t0, addresses ldl_t1
WHERE (ldl_t0.locCode='2223525024894')
        AND (ldl_t1.telNumber=ldl_t0.telNumber)
        AND (ldl_t1.adrs_type='BLG')
OPEN LdlEssQueryCursor1
FETCH LdlEssQueryCursor1
<< PO BOX F          MYTOWN      ST   USA
FETCH LdlEssQueryCursor1
CLOSE LdlEssQueryCursor1

FETCH LdlEssQueryCursor0
<< 2224400859056

DECLARE LdlEssQueryCursor1 CURSOR FOR
SELECT DISTINCT ldl_t1.address
FROM numbers ldl_t0, addresses ldl_t1
```

```
WHERE (ldl_t0.locCode='2224400859056')
        AND (ldl_t1.telNumber=ldl_t0.telNumber)
        AND (ldl_t1.adrs_type='BLG')

OPEN LdlEssQueryCursor1
FETCH LdlEssQueryCursor1
<< 5525 CENTRAL AV   BIGCITY     ST   USA

FETCH LdlEssQueryCursor1
CLOSE LdlEssQueryCursor1

FETCH LdlEssQueryCursor0
2224403316619

...  This iterates through every element in the Locations table.

FETCH LdlEssQueryCursor0
CLOSE LdlEssQueryCursor0
```

To take advantage of the SQL aggregate function, this query should be written as:

```
multipleAddrs(LocCode, N)<-
        numbers(LocCode, _),
        addrs(N, 'BLG', LocCode),
        N > 1.

addrs(count<Address>, Type, LocCode) <-
        numbers(TelNumber, LocCode),
        addresses(TelNumber, Type, Address).
```

Which generates the following SQL statements:

```
DECLARE LdlEssQueryCursor0 CURSOR FOR
SELECT DISTINCT ldl_t0.locCode, count(ldl_t2.address)
FROM locations ldl_t0, numbers ldl_t1, addresses ldl_t2
WHERE (ldl_t1.locCode=ldl_t0.locCode)
        AND (ldl_t2.telNumber=ldl_t1.telNumber)
```

```
        AND (ldl_t2.adrs_type='BLG')
GROUP BY ldl_t0.locCode HAVING (count(ldl_t2.address)>1)

OPEN LdlEssQueryCursor0

FETCH LdlEssQueryCursor0
FETCH LdlEssQueryCursor0
FETCH LdlEssQueryCursor0
...
CLOSE LdlEssQueryCursor0
```

This only fetches the LocCodes for tuples that have multiple addresses, and avoids the performance penalty of actually copying all of the items from the source tables into the LDL++ system.

6 CONCLUSIONS

We discovered that our choice of the LDL++ system for the deductive engine for data validation and cleanup tasks was appropriate — in fact we plan to use it to address real world problems in the near future. Two key advantages of the LDL++ system to meet our requirements are:

- The ability to express data validation and cleanup rules in LDL, Prolog, and C++ languages can be used for complementary purposes. the LDL++ system enables the rule developer to pick the language and implementation most suited for the particular task at hand.
- Data lives in commercial DBMSs. For our application, the LDL++ system has an advantage over many other deductive database systems because it can validate and clean data managed by commercial DBMSs in currently deployed systems.

Our experience of using deductive database technology, in particular LDL, showed that most useful applications of this technology will require that it be coupled with other technological components, especially user interface components and commercial database management systems. It is unlikely and also undesirable that all three components be deployed on the same computer system, so issues of open and efficient distributed computing and client/server

technologies also come into play. Our key conclusion is that for deductive database technology to gain wider acceptance and usage in the real world, it must be integrated well with these technologies, especially the distributed computing technology. Furthermore, this will involve meeting both research and engineering challenges to achieve better integration.

To meet the requirements of a practical interactive application based on deductive database technology, we believe that deductive database technology will have to be extended in two crucial aspects.

- Operational aspects

  This refers to the aspects related to configuring various processes on different computers to meet performance requirements. Configuration choices, such as the machines on which the the LDL++ system server and the user interface are executed, become important.

- Data definition aspects

  To support better optimization of query processing, the deductive database system needs to take into account and make design choices related to the following factors:

  - The location and the size of various tables involved, to choose a good query execution plan.
  - Determining what data access methods (indices, clusters, etc.) exist, and (re)creating these structures if they are needed to improve query performance. This has previously been the domain of database administrators (DBA). However, deductive database systems make the relationship between queries and access structures needed to efficiently evaluate the queries much less clear than with traditional query languages such as SQL. Compiled deductive database systems have the knowledge needed to advise a DBA, at compile time, where indices and clustering are needed for adequate performance of the queries.
  - Migrating tables so that join operations can be performed within DBMSs, instead of forcing the rule processing component to perform the join operation.

Many of the above performance issues have been researched in distributed database and multidatabase systems contexts. We also observed that the optimization capability of the LDL++ system with respect to these issues is rather

limited. We believe that the data definition capabilities of a deductive database language like LDL should be enhanced to capture the above information. The other alternative is to develop a mapping between LDL and a multidatabase language like MSQL[19]. We may then be able to depend on the data definition capabilities of MSQL to capture the above information and use multidatabase query processing strategies to achieve better performance.

Acknowledgements

Our effort to develop Q-Data would not have been possible without the support of KayLiang Ong and Christine Tomlinson at MCC. Our effort to trial Q-Data would not have been effective without the help of several colleagues in Bellcore's client companies, including Rodolphe Nassif and Jian-Hua Zhu, who discussed real problems and provided real data.

REFERENCES

[1] A. Natraj and K-L. Ong, "The LDL++ User's Guide, Edition 2.0," MCC Technical Report Carnot-012-93(P), January 1993.

[2] C. Baker, H. Lu, K. Mikkilineni, R. Ramnarayan, J. Richardson, A. Sheth, and S. Yalamanchili, "Very Large Parallel Data Flow," Final Report F30602-85-C-0215, Honeywell Corporate Systems Development Division, December 1987.

[3] "The Bellcore OSCA Architecture", Bellcore Technical Report TR-STS-000915, October 1992.

[4] "The LDL System Prototype", IEEE Transactions on Knowledge and Data Engineering, 2 (1), March 1990.

[5] C. Fox, A. Levitin, and T. Redman, "The Notion of Data and its Quality Dimensions," Information Processing and Management, 30 (1), 1994, pp. 9-19.

[6] Y. Ioannidis and M. Tsangaris, "The Design, Implementation, and Performance Evaluation of BERMUDA," IEEE Transactions on Knowledge and Data Engineering, 6 (1), February 1994, pp. 38-56.

[7] G. Karabatis and A. Sheth, "Multidatabase Interdependencies in Industry," (Industrial Session), Proc. of the ACM SIGMOD'93, May 1993.

[8] S. Madnick, "The Voice of the Customer: Innovative and Useful Research Directions," (Panel Position Statement), Proc. of the 19th VLDB, August 1993.

[9] S. Naqvi and S. Tsur, A Logical Language for Data and Knowledge Bases, Computer Science Press, New York, 1989.

[10] R. Nassif and D. Mitchusson, "Issues and Approaches for Migration/Cohabitation between Legacy and New Systems," (Industrial Session), Proc. of the ACM SIGMOD'93, May 1993.

[11] A. O'Hare and A. Sheth, "The Interpreted-Compiled Range of AI/DB Systems," SIGMOD Record, 18 (1), March 1989.

[12] B. Patterson, "The Need for Data Quality," (Panel Position Statement), Proc. of the 19th VLDB, August 1993.

[13] G. Phipps, M. Derr, and K. Ross, "Glue-NAIL!: A Deductive Database System," Proc. of ACM SIGMOD'91, May 1991.

[14] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri, "Implementation of the CORAL deductive database system," Proc. of the ACM SIGMOD'93, May 1993.

[15] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri "The CORAL Deductive System", The VLDB Journal, Volume 3(2), April 1994.

[16] S. Shaw, L. Foggiato-Bish, I. Garcia, G. Tillman, D. Tryon, W. Woods, C. Zaniolo, "Improving Data quality via LDL++," In Proc. of ILPS 93 Workshop on Programming with Logic Databases, November 1993.

[17] A. Sheth and A. O'Hare, "The Architecture of BrAID: A System for Bridging AI/DB Systems," Proc. of the 7th Int'l. Conf. on Data Engineering, April 1991.

[18] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan, "Coral++: Adding Object-Orientation to a Logic Database Language," Proc. of the 19th VLDB, August 1993.

[19] W. Litwin. MSQL: A Multidatabase Language. *Information Sciences*, 1990.