Extending Data Processing Capabilities of Relational Database Management Systems.

Igor Wojnicki
University of Missouri – St. Louis
Department of Mathematics
and Computer Science
8001 Natural Bridge Road
St. Louis, MO 63121–4499
phone: (314) 516-6353
fax: (314) 516-5400
e-mail: wojnicki@arch.umsl.edu

Cezary Z. Janikow
University of Missouri – St. Louis
Department of Mathematics
and Computer Science
8001 Natural Bridge Road
St. Louis, MO 63121–4499
phone: (314) 516-6352
fax: (314) 516-5400
e-mail: janikow@arch.umsl.edu

Abstract *Relational Database Management Systems proven to be robust and efficient for storing and retrieving data. However, they have limitations. Sophisticated data processing (a rule-based processing with inferences) is possible but hard to implement. To overcome these problems, several methods have been developed. Data can be processed by a client application (a client-side processing) or by a RDBMS server (a server-side processing). These methods overcome the main limitations, but such solutions are not flexible, hard to modify or alter. This paper presents a novel approach, a case of server-side processing with the use of a declarative language. A program, written in Prolog, is decomposed into relations, and stored in the database. It can be easily managed, modified or altered. It has access to database relations, and it also has an ability to create relations on demand, so-called dynamic views or* Jelly Views. Jelly Views *serve as temporary relations holding data obtained as the result of the program. The communication interface between the client and the database is preserved, it is still SQL.*

Keywords: RDBMS, Prolog, deductive database, inference

1 The Relational Model

The relational model represents a database as a collection of relations [1, 2, 3]. Each relation is characterized by its name and attributes. A set of attribute values for a certain relation is called a tuple. A value of the attribute is determined by the attribute domain (data type). A domain is a set of atomic (indivisible) values. A relation schema is defined as a relation name (R) and a list of attributes(A_i):

$$R(A_1, A_2, \ldots, A_n)$$

The domain is defined for each attribute $dom(A_i)$. A relation state (or just relation) of the relation schema $R(A_1, A_2, \ldots, A_n)$, denoted by $r(R)$, is a set of tuples $t_1, \ldots, t_m$, where each tuple is an ordered list of n values $t =< v_1, \ldots, v_n >$, and each value v_i is an element of $dom(A_i)$ or $null$ (does not exist). The values in tuple t are referred to as $t[A_i]$, so by the attribute's name. Relations represent facts about entities or facts about relationships between relations. A relational database schema is a set of relation schemas: $S = R_1, \ldots, R_o$

2 RDBMS Boundaries

Each Relational Database Management System uses two languages: Data Definition Language for defining a relational database schema and Data Manipulation Language for storing and retrieving data.

SQL has evolved as a high level, robust and compact DDL and DML query language. The power of SQL lies in its declarative nature. A query can be expressed in a logical form with little care for procedural processing. It gives meaningful and highly abstractive way for querying. But here is a limitation. SQL offers an abstract interface allowing to build sophisticated queries, but more complicated data processing is out of reach. In particular, the following disadvantages can be identified:

- lack of more sophisticated, i.e. rule-based data processing,
- a query can not be recursive,
- searching for acceptable solutions is limited.

Rule-based processing requires rules, usually the a form of decision trees. Decisions are based on information acquired from a database. At each node of such a tree, the database is queried and the replay is compared with appropriate values (or set of values) in the node. Depending on the comparison, the next node of the tree is taken and the process continues. Rule-based processing is a core part of the most Expert and DDB (Deductive Database) systems [4, 5, 2].

Recursive queries, in general, regard searching for data which is stored in some tree-like or graph-like way. The Traveling Salesman Problem can be an example. A relation holds information about distances between neighboring cities, there is no data redundancy. A relaxed question is: find the shortest path between two given cities. If the question was asked with known number of cities on the path, it would be issued as an ordinary query. However, usually the number of cities is not known in advance. SQL3 (known also as SQL99) standard allows recursive queries. However, this feature is very rarely implemented (it is implemented in DB2 [6]).

The *Acceptable Solutions* problem is another application where RDBMS can not be applied directly. It can be also called a *Reverse Aggregation*. Let's use the same Traveling Salesman Problem. The question would be: find all paths shorter then some value. Each path is a list of cities. As a result, the replay would be a set of paths. Each path would be shorter than the given in the query value.

There are several applications where these problems appear:

- phone billing: this usually involves sophisticated rule-based processing, cost of connections is calculated basing on different variables: day time, number of active phone numbers of the customer, calling plan, length of the call etc.; rules are subject to change,
- flight planes, trip planning (the Traveling Salesman Problem in general),
- budget planning, alternative flight/trip routes.

These problems can be tackled using either client-side processing, or server-side processing with functions.

Client-side processing is based on serialization of queries and taking actions accordingly to partial replies. The process is as follows: the client sends a query, the RDBMS replies, depending on the reply the client sends another query. The logic is usually hard-coded into the client. There are a few drawbacks of such a solution:

- the inference process involves many queries,
- queries are not optimized by the RDBMS (series of simple queries and travel through a decission tree instead of few more complex queries),
- with the decision tree hard-coded into the client, it is hard to modify the logic,
- only dedicated client software is able to conduct the inference process, and thus

the principle of a database as a universal source of information fails.

Such a solution is implemented in expert systems, with an external RDBMS data source.

Server-side processing seems to be more robust, but there is a need for support from RDBMS in the form of additional languages, usually procedural ones. Using such a language, a function can be written. A user query can use such a function as a data-source. Comparing to client-side processing, the number of queries is reduced, a function is stored and processed by the RDBMS so it is optimized, and the database remains an universal source of information. Still, there are a few drawbacks:

- the function's language is RDBMS specific, can be used only for dedicated RDBMS,
- logic, i.e. decission tree, is hard-coded within the function code, inflexible, hard to modify,
- the input and output for the function have to be known when the function is defined.

3 Relational Model vs Logic

In general, knowledge processing involves two kinds of data: extensional and intensional.

Extensional knowledge is just bare facts, while intensional defines relationships among facts. Logic can be used to express extensional and intensional knowledge as well [7]. To formalize syntax for writing logical sentences, the alphabet has to be stated. Data (or knowledge) is stored as predicates. A predicate p is defined as a symbol denoting appropriate relation among n number of arguments:

$$p/n$$

where n is called its *arity*. A fact compound of n values, concerning predicate p is named a simple clause:

$$p(a_1, \ldots, a_n)$$

where, $< a_1, \ldots, a_n >$ is a tuple (ordered list of values). There is an analogy between a relation and a clause. (in terms of the Relational Model see Sec. 1) Both of them describe facts using tuples. In the case of the Relational Model, a value in the tuple can be referenced using appropriate attribute name or its position, while for the Logic Programming only position can be used. Concluding, any Relational Model tuple can be represented as a simple clause, so compatibility between the relational databases and logic programs has been stated.

Furthermore logic programs can be constructed with so-called complex clauses. Such a clause doesn't express facts explicitly, but defines relationships among them (intensional knowledge), with a conclusion.

$$\begin{aligned} con(C_1, \ldots, C_n) \quad \leftarrow \quad & c_1(C_i \ldots, C_j) \triangle_1 \ldots \\ & \ldots \triangle_r \; c_m(C_k, \ldots, C_l). \end{aligned}$$

Here, *con* is the concluding clause with arity n; $\triangle$ are logical operators joining predicates c_q (preconditions). Predicates such as c_q are matched against their clauses. If such a match is found, appropriate facts are set and the concluding tuple is passed as a new fact. As a result, a non-deterministic conclusion can be drawn; a single complex clause can generate multiple conclusions, and new facts. Functionality of complex clauses is not supported by the Relational Model.

Logic Programs provide a uniform language for representation of databases and even more, enabling additional expressive power in the form of intensional knowledge.

4 Logic Programs in RDBMS

The main proposed improvement to RDMBS is adding capabilities to store and process intensional knowledge, along with the extensional one. The inference engine will be coupled with the data source, and available to a client through standard SQL queries [8]. This approach addresses the major disadvantages of current RDBMS (see Sec. 2): limited rule-based processing, recursive queries, searching for acceptable solutions.

To keep the architecture flexible and easy to modify, the following assumptions are taken:

- logic programs are used to construct a view called a *Jelly View*,
- the view is created on demand,
- the query language is preserved, the view is transparent – not different from other RDBMS views.

As a result, RDBMS has its functionality extended toward those of deductive databases (DDB).

The chosen syntax is that of the Prolog language. This provides all necessary mechanisms for declaring intensional knowledge, and it also provides some features of procedural languages that allow to control the inference process. In turn, this allows to create logic programs and optimize them. A Prolog language program is similar to pure logical programs, with some technical differences [9, 7] (see Sec. 3). To accomplish the coupling of the Prolog program with RDBMS, that program must be represented in a way complying with the Relational Model, including the normal forms. There are following challenges to face:

- *Internal Matching* — matching between relations and clauses, enabling access to data (extensional knowledge) from the logic program,
- *External Matching* — stating the goal for the prolog program to generate a *Jelly View*,
- *Logic Program* itself, which should be decomposed to meet the normal form requirements.

These challenges can be met by the three main parts of a Prolog program: extensional knowledge (*Internal Matching*), intensional knowledge (*Logic Program*) and goal (*External Matching*).

Let's define a behavior of the system. All queries requiring a *Jelly View* have to be processed. Once such a query appears (detected by *External Matching* routines), the *Logic Program* should be built. All extensional data is available to the *Logic Program* through the *Internal Matching* mechanism. Then the inference engine generates the necessary *Jelly View*. Upon the completion of the query, all dynamic data (the *Jelly View*) is assumed to be no longer valid and thus it is removed. If a query doesn't refer to *Jelly View*, it is passed on to the RDBMS, and the response is passed to the user. In other words, the system in transparent to the user.

5 Decomposition

Decomposing the three main components of a *Jelly View*: *External Matching*, *Internal Matching* and *Logic Program*, is a crucial problem. They must comply with the relational model. One possible approach is illustrated in the ER diagram in Fig. 1.

External Matching matches a relation name and a clause name. If a relation name is found in `table-clause.table`, a *Jelly View* for this relation is generated. The *Jelly View* has attributes defined by the entity `argument`. To generate the view, the clause name is specified (`table-clause.clause`) and subsequently used as goal for the inference engine. The arity of the goal predicate is calculated from the number of arguments (the relation `argument`).

Internal Matching is a bridge between the inference engine and the RDBMS, it handles extensional knowledge. A predicate can be mapped on a relation. If the inference engine requests a simple clause dealing with the predicate named `clause-table.name` of arity `clause-table.arity`, the clause is generated from the data stored in the relation `clause-table.table`. This is the way extensional knowledge stored in the RDBMS is accessed by the inference engine.

The goal for the logic program has to be specified. It is held by the `clause` relation.

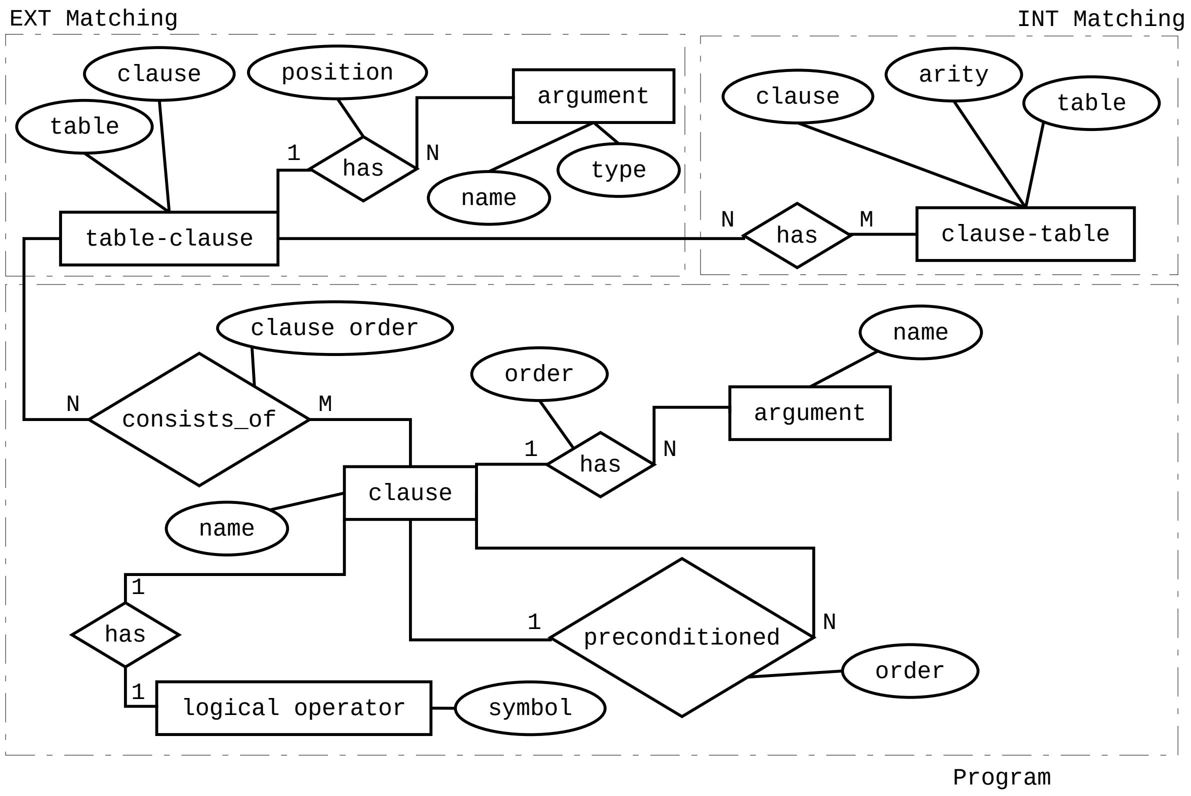

Figure 1: External Matching, Internal Matching and Logic Program Entity Relationship Diagram.

Each logic program consists of certain number (one or more) of complex clauses. Each `table-clause` is an ordered list of complex clauses (the relation `clause`). A single clause has a number of named arguments. This clause is either a concluding clause or a precondition predicate. These are not differentiated because they are represented by the same relation. As in Prolog, such a construct always has a logical operator on the right (the relation `logical operator`). The concluding clauses and precondition predicates are not distinct in terms of decomposition., `preconditioned` relationship is used to assign preconditions to the concluding clause. As shown, any complex clause defined in Sec. 3 can be decomposed into such entities.

6 Conclusions

Extending knowledge processing capabilities of RDBMS makes them applicable to new domains and situations. Combining speed and efficiency of a RDBMS with a server-side declarative programming gives a powerful, robust, and flexible environment, not only for data storage and retrieval, but also for sophisticated processing such as drawing conclusions and discovering new knowledge. A client application issues ordinary SQL queries, all the processing is done on the server side. Intensional knowledge (rules) is stored within RDBMS as a decomposed Prolog program. This architecture makes the system flexible and transparent to the client. Any rule can be altered individually, with changes applied only to appropriate relations, using ordinary SQL queries. This can be done without altering any client application. As a result, RDBMS can have DDB functionality, preserving communication methods and the Relational Model.

This approach can have applications in many fields of Computer Science, including:

- Data Mining and Data Discovery,

- rule systems, e.g., telephone billing systems, tax systems,
- Expert Systems, including those self-teaching (RDBMS could now posses Expert System capabilities),
- Artificial Intelligence, Knowledge Discovery tools.

References

[1] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[2] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2000.

[3] Jeffrey D. Ullman and Jennifer Widom. *A first course in Database systems*. Prentice-Hall Inc., 1997.

[4] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. The glue-nail deductive database system: Design, implementation, and evaluation. *VLDB Journal*, 3(2):123–160, 1994.

[5] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *VLDB Journal: Very Large Data Bases*, 3(2):161–210, 1994.

[6] Srini Venigalla Netsetgo. Expanding recursive opportunities with sql udfs in db2 v7.2. Technical report, International Business Machines Corporation, 2002.

[7] Ulf Nilsson and Jan Małuszyński. *Logic, Programming and Prolog*. John Wiley & Sons, 1990.

[8] Igor Wojnicki and Antoni Ligęza. An inference engine for rdbms. In *6th International Conference on Soft Computing and Distributed Processing*, Rzeszów, Poland, 2002.

[9] Michael A. Covington, Donald Nute, and André Vellino. *Prolog programming in depth*. Prentice-Hall, 1997.