Computer Science • Vol. 6 • 2004

Igor Wojnicki*

JELLY VIEWS: EXTENDING RELATIONAL DATABASE SYSTEMS TOWARD DEDUCTIVE DATABASE SYSTEMS

This paper regards the Jelly View technology, which provides a new, practical methodology for knowledge decomposition, storage, and retrieval within Relational Database Management Systems (RDBMS). Intensional Knowledge clauses (rules) are decomposed and stored in the RDBMS founding reusable components. The results of the rule-based processing are visible as regular views, accessible through SQL. From the end-user point of view the processing capability becomes unlimited (arbitrarily complex queries can be constructed using Intensional Knowledge), while the most external queries are expressed with standard SQL. The RDBMS functionality becomes extended toward that of the Deductive Databases.

Keywords: *RDBMS, Prolog, Intensional Knowledge, Deductive Database, Recursive Queries*

JELLY VIEWS: ROZSZERZENIE RELACYJNYCH BAZ DANYCH W KIERUNKU DEDUKCYJNYCH BAZ DANYCH

Niniejsza publikacja prezentuje technologię Jelly View, która udostępnia nową, praktyczną metodologię dekompozycji, przechowywania i przetwarzania wiedzy w Systemach Zarządzania Relacyjnymi Bazami Danych (SZRBD). Klauzule Wiedzy Intensjonalnej (reguły) przechowywane są w SZRBD w formie zdekomponowanej, tworząc modularne komponenty. Rezultaty przetwarzania regułowego reprezentowane są jako widoki, których stan dostępny jest za pomocą zapytań SQL. Z punktu widzenia końcowego użytkownika możliwości przetwarzania danych stają się nieograniczone (wykorzystując reguły), podczas gdy zapytania wyrażane są wciąż w języku SQL. Jako rezultat, funkcjonalność SZRBD zostaje zwiększona do poziomu funkcjonalności Dedukcyjnych Baz Danych.

Słowa kluczowe: *SZRBD, Prolog, Wiedza Intensjonalna, Dedukcyjne Bazy Danych, Zapytania Rekurencyjne*

1. Introduction

The principal limitations of the Relational Database Management Systems (RDBMS) can be identified as: lack of more sophisticated, i.e. rule-based data processing (**C0 class**), traversal of structurally complex data structures (such as graphs, trees, terms, lists etc.) (**C1 class**) and search for Admissible Solutions under specified constraints

*Institute of Automatics, AGH University of Science and Technology; Department of Mathematics and Computer Science, Univeristy of Missouri – St.Louis; e-mail: `wojnicki@agh.edu.pl` (finding specific subsets of a given set, generation of structural solutions satisfying specific constraints etc.) (**C2 class**). The classes **C1** and **C2** follow from a generic drawback of relational databases which is the *difficulty handling recursive queries* [2]. Such a difficulty is one of the eight principal weaknesses of Relational Database Management Systems discussed in [2]. The most important generic problem following this difficulty is the one mathematically defined as finding *transitive closure* of relation. Apart from some small-scale examples this operation is computationally infeasible due to its complexity. However certain specific cases can be solved even for practically useful systems. The class **C0** is more general, and it might be a superset of **C1** and **C2**. Providing a solution to **C0** which allows recursive processing applies also to **C1** and **C2**.

The main topic this paper is focused on development of a methodology called *Jelly View*, which embeds Logic Programming into Relational Databases with some minimal overhead concerning software technology. Such an approach enables more spohisticated processing within Relational Database Management Systems, like that of Deductive Database Systems. All the data and knowledge necessary for computations are stored using the Relational Database paradigm; no external knowledge base is required. The difference between *Jelly View* and Deductive Database Management System (DDBMS) is that it extends existing Relational Database, preserving its features, including SQL as the communication language. *Jelly View* extends the RDBMS *System Catalog*[1] toward storing Intensional Knowledge (rules) in a form of *Logic Programs*.

The proposed methodology is supported with a technology which provides dynamic generation and evaluation of *Logic Programs* on demand. It is implemented as an external, loosely coupled, module independent of specific RDBMS; this makes the technology extremely portable and applicable to practically any database system. The inference capability is provided by integrating a PROLOG rule-based inference engine with the database. The results of the inference process are perceived as regular database relations (views) by the end-user and they are accessible through SQL.

The *Jelly View* technology introduces inference capabilities which enhance the functionality of Relational Database Systems which allows solving **C1** and **C2** classes of problems, including recursive processing. It also provides more general capabilities, which is rule-based processing (**C0 class**).

2. Relational Databases vs. Deductive Databases

The Relational model was introduced for the first time as "A Relational Model of Data for Large Shared Data Banks" in 1970 [1]. Perhaps one of the most important reasons making the model so popular, is the way it supports powerful, yet simple

[1]It is where the relational systems store information about databases, tables, columns, etc., some systems call it the *Data Dictionary*; the Catalogs appear to the user as tables like any other, but the RDBMS stores its internal bookkeeping in them.

declarative languages. These languages express operations which can be applied to data [10]. The Relational Data Model operations are defined by Relational Algebra, these are [6]: Union, Difference, Cartesian Product, Projection, Selection.

The Relational Model provides a way to store and process data. But still, there is a need for a language to serve as a uniform communication method between the user and the database back-end. Such a language has to provide methods to insert, extract and alter data, gathered in the database.

There are two basic issues to cover: data definition and data manipulation. *Data Definition Language* has a capability of defining a relational database schema, while *Data Manipulation Language*, of storing and retrieving data. A database system, having DDL and DML interpreter and a physical back-end, which comply with the Relational Model, is called the *Relational Database Management System.* The most wide spread DDL and DML query language is SQL [6, 5, 2, 11, 10].

Deductive Databases (DDB) are conceptual extension of Relational Databases which supports more complex data modeling. In general, they are logic programming systems designed for applications with large quantities of data. Deductive databases generalize relational databases by exploiting the expressive power of (potentially recursive) logical rules and of non-atomic data structures [13]. Such approach greatly simplifies the task of application programmers, providing extended knowledge processing capabilities at the database level. The database becomes not only a data-source, but also a knowledge-source. Thus, a deductive database is a combination of a conventional database containing facts, knowledge base containing rules, and inference engine which allows the derivation of information implied by the facts and rules.

There have been several efforts taken to create such systems [10, 5]. In general, a deductive database system is based on the two following principles [10]:

1) It has a declarative language, that is logic, serving both as the communication method (query language) and a host language, providing DDL and DML.
2) It supports principle features of database systems, that is efficient access to massive amounts of data, sharing of data, and concurrent access to data.

At the beginning of twenty first century, there are no production systems of this nature available. However, there are many experimental implementations like: CORAL [9], Glue-NAIL! [4], the Aditi Project [12], the LDL System [5], Lack of popularity of such systems is usually caused by non-standard languages used as a communication method between the user and the database.

Most of the contemporary database systems are based on the Relational Model of Data, thus SQL. There are many well tested ways of query optimization for SQL, and many applications that use it, not mentioning the number of qualified SQL, or in general, database programmers. SQL is designed for the Relational Model. Since Relational Algebra handles data only, it does not anticipate a need for intensional knowledge, nor does SQL.

There are some attempts to increase SQL expressive power towards this of DDB. The most significant one is an extension which allows recursive queries. It is constituted by SQL99, but it has not been implemented in many RDBMS. Actually, the only RDBMS that supports it, is DB2 by IBM [7].

The distance from enabling recursive queries to intensional knowledge processing is very long, and basically it is not possible to make SQL support it without some major redesign. A major redesign causes major changes and presumably incompatibilities regarding already running RDBMS. These incompatibilities prevent the changes. No company can afford such a major switch from one technology to another.

And that is why *Jelly View* approaches the problem from a different perspective. It extends the *System Catalog* making the database "aware" of intensional knowledge, not changing the communication language.

3. Extending Relational Database Management System

The primary goal to achieve is to overcome the limitations of RDBMS mentioned in Section 1. In general, it can be done by introducing intensional knowledge, a set of rules, which may be applied to extensional knowledge already gathered in the database. Furthermore, there is a need for an engine, which will interpret the rules and provide the inference.

Following these guidelines there are the following requirements:

- keep intensional knowledge in a well known, well tested, and efficient way in the RDBMS,
- allow declarative programming, so the inference will be described by logical rules,
- allow controlling of the inference process, incorporating procedural programming elements with respect to the previous statements,
- minimize use of additional programming languages and paradigms: be as close to the foundations of RDBMS as possible, so database designers and programmers can adjust easily.

The main proposed improvement to RDMBS is providing capabilities to store and process intensional knowledge (rules), along with the extensional one. Meta-data regarding rules is stored as an extension of the System Catalog. Furthermore, the Prolog inference engine is coupled with the database, and available through standard SQL queries [16, 15, 14]. This approach addresses the major disadvantages of current RDBMS and it allows to solve problems from classes: **C0**, **C1** and **C2**, since they are solvable in Prolog.

Prolog also provides some procedural extensions [3]. They allow to control the inference process more strictly and provide procedural programming where it is needed. By default, Prolog offers backward-chaining [3]. But as it is proven in [3, 8], it may be used as a forward-chaining inference engine, as well. Some rules may serve as a modification of the existing backward-chaining engine, providing forward-chaining functionality, while other rules are a meta-program for the forward-chaining inference engine.

Prolog is based on predicate logic [3, 8], so it is very closely related to the foundation of the relational model. Using Prolog, there is no need for thinking in a non-relational way. If one is familiar with the relational model, he or she is also familiar with the logic behind it. Then he or she is also familiar with Prolog, with some minor adjustment concerning the syntax, at most.

Summarizing:

- Extensional knowledge is provided by the database; it is covered by relations.
- Intensional Knowledge is expressed by Prolog programs.
- A Prolog program is decomposed and stored in RDBMS,
- The Prolog program consists of an ordered set of rules[2] named clauses.
- The decomposition process is well-defined; meta-data regarding intensional knowledge extends the System Catalog.
- Inferred data is perceived as a dynamically generated view, called *Jelly View*.
- State of the *Jelly View* is generated as a result of the inference process targeted by a goal over some number of clauses.

As a result, RDBMS has its functionality extended toward those of Deductive Databases.

4. Flexibility and Reusability of Decomposed, Modularized Intensional Knowledge

Intensional knowledge is stored into data as decomposed Prolog clauses. The ER diagram visualizing the decomposition is given in Figure 2, section `Program`. The program is targeted by the goal which is given as a predicate denoted by `predicate` and `arity`. Each program consists of some number of ordered clauses. The order of clauses is provided by `clause order`.

Each complex clause is composed of the head and the body. The head is a predicate name, denoted as `name` (`clause` entity) and a list of parameters, denoted as `argument` entity. Each of the parameters has a `name` and `position`.

The body consists of some number of ordered subgoals (`preconditioned` relationship). Each subgoal is described by `clause` entity. The subgoals are separated by logical operators (`logical operator` entity), which are a comma (’,’) or a semicolon (’;’). Additionally, there should be two more operators defined: an implication (denoted as ’:-’, which is equivalent to ←) to separate the head and the body, and a period (’.’) at the end of the complex clause.

For example, having the following complex clause:

```
sibling(X,Y) :- parent(X,Z), parent(Y,Z), X \== Y.
```

[2]The order makes difference when it comes to inference and if the inference process is to be controlled or altered.

It is decomposed into the relations given in Table 1. The decomposition process is visualized in Figure 1.

Table 1
A decomposed clause and its description

`clause`

id	name	order	preconditioned	symbol
1	sibling	NULL	NULL	1
2	parent	1	1	2
3	parent	2	1	2
4	\==	3	1	4

`argument`

name	position	clause
X	1	1
Y	2	1
X	1	2
Z	2	2
Y	1	3
Z	2	3
X	1	4
Y	2	4

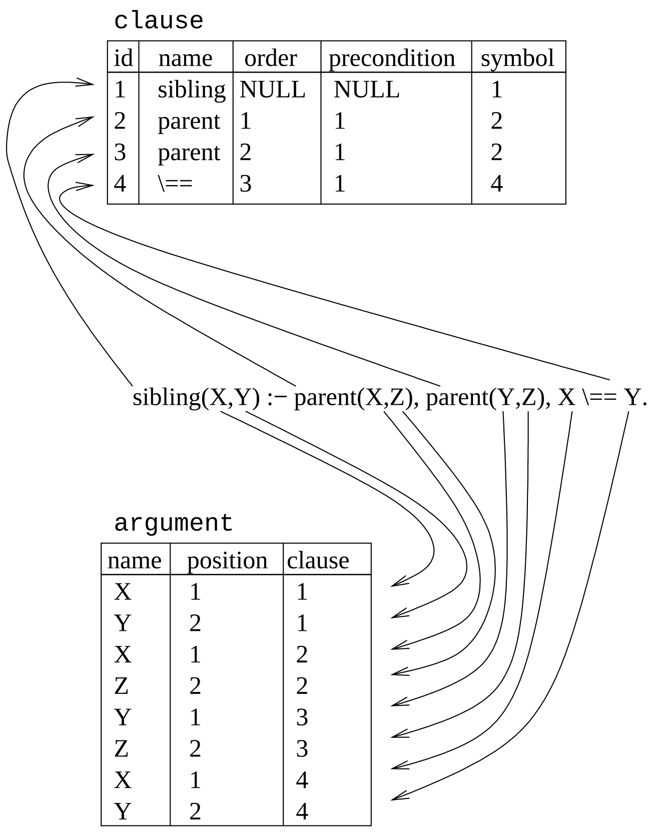
`logical operator`

id	symbol
1	`:-`
2	`,`
3	`;`
4	`.`

attribute	description
`clause.id`	primary key for `clause`
`clause.name`	name of the predicate in the head or body
`clause.order`	an arbitrary integer denoting the order of predicates in the body
`clause.preconditioned`	the foreign key referring to `clause.id`, if not `NULL` it means that the predicate is in the body of a clause referred by it
`clause.symbol`	the foreign key referring to `logical operator.id`, the logical operator at the end of predicate
`argument.name`	name of the predicate argument
`argument.position`	position of the argument in predicate, an arbitrary integer
`argument.clause`	foreign key referring to the predicate `clause.id`
`logical operator.id`	primary key for `logical operator`
`logical operator.symbol`	the operator symbol

The program could be decomposed further covering non-atomic values of parameters: structures and lists. But such a parameter decomposition will require a recursive query, to follow all the subsequent parameters: parameters which are structures, which have parameters which are structures, and so on.

It may have tremendous impact on the performance[3]. This is the reason why predicate parameters are not subject to further decomposition. Summarizing: constants, variables, structures and lists are all perceived as atomic arguments, so there will be a single record for a single parameter.



`clause`

id	name	order	precondition	symbol
1	sibling	NULL	NULL	1
2	parent	1	1	2
3	parent	2	1	2
4	\==	3	1	4

sibling(X,Y) :− parent(X,Z), parent(Y,Z), X \== Y.

`argument`

name	position	clause
X	1	1
Y	2	1
X	1	2
Z	2	2
Y	1	3
Z	2	3
X	1	4
Y	2	4

Fig. 1. Decomposing a clause

The decomposition resembles catalog-driven approach present in the most of the relational systems. The relational systems store information about databases, tables, columns, etc., in *System Catalogs* (Some systems call this the *Data Dictionary*). The Catalogs appear to the user as tables like any other, but the RDBMS stores its internal bookkeeping in them. The decomposed logic program may be perceived as an extension of the regular system catalogs, which allows to store intensional knowledge. It is just a step forward comparing with the *system catalog* of contemporary RDBMS.

[3]This topis is subject to further research.

In order to provide complete meta-data regarding *Jelly Views* the follwing issues have to be addressed:

- the name of the *Jelly View* has to be stated for each decomposed PROLOG program,
- the *Jelly View* has to have a description of its parameters in terms of database: column names and their data types,
- the PROLOG program (thus, the inference engine) should have access to relations in the database which provide extensional knowledge.

The first two items are covered by so-called *External Matching*. The third one is covered by the *Internal Matching*.

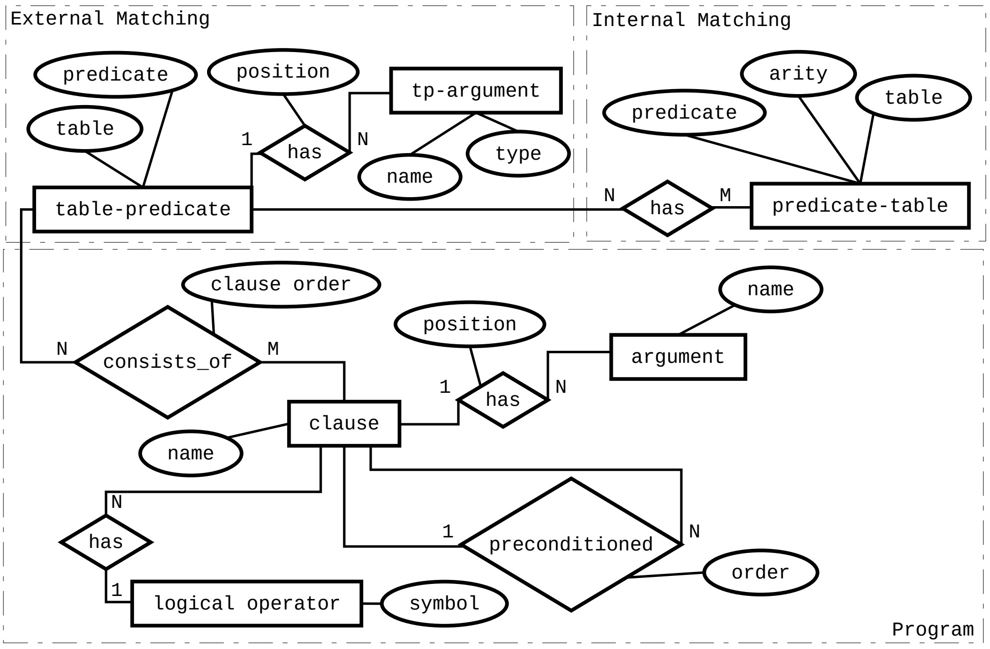


Fig. 2. External Matching, Internal Matching and Logic Program, Entity Relationship Diagram

The *External Matching* provides relationships between the *Jelly View* name, its parameters in terms of database, and the program. The *Internal Matching* provides relationships between predicates used by the program, and the relations. The inference engine becomes a data source for the database by the *External Matching*, and the database becomes a data source for the inference engine by the *Internal Matching* (see Fig. 2).

The `External Matching` section in Figure 2 establishes a relationship between the database and the inference engine. A name of the *Jelly View* is listed as `table-predicate.table`. In order to generate the data, an appropriate goal will be called which is a predicate name in `table-predicate.predicate`. The number of arguments of the *Jelly View* matches the arity of goal, which is (`N`). The column names and their data types are provided by `tp-argument` relation.

The `Internal Matching` section in Figure 2 provides access to the database from the inference engine. It maps database relations into PROLOG predicates. The mapping is provided by `predicate-table` relation. If there is a need to access a relation, then its name is listed in `predicate-table.table` and a corresponding predicate name is listed as `predicate-table.predicate`. The inference engine gets access to the relation using the predicate name. The arity of the predicate is indicated as `predicate-table.arity`, which is also the number of columns in the relation.

The clauses (`Program` section in Fig. 2) are binded to the *Jelly View* by `consist_of` relationship. Each program has some number of internal matching mappings, which are provided by `has` relationship between `table-predicate` and `predicate-table`. In this way, clauses and predicate-to-table mappings can be reused in different programs. Two or more programs can share some of the clauses and some of the Internal Matching mappings, then.

Summarizing, a Logic Program is decomposed into relations which results in the following benefits:

- Programs become modular, clauses can be reused in different programs.
- Clauses can be easily modified, added or removed, using SQL.
- The structure of clause is given explicitly which enables further formal analysis of theoretical properties such as *completeness*.

Regarding the modular design. Programs are decomposed at the clause level. The same clause may be shared by more than one program, therefore it can be reused.

There are the following programming and usage scenarios available then:

- The user can write an entire PROLOG program to create a *Jelly View*, and query it.
- The user can assemble the program from the existing clauses to create a new *Jelly View*, and query it.
- The user can query an existing *Jelly View*.

Concerning easy modification. The program is decomposed and stored into relations. SQL is used to add, remove or alter the program then. The same method is used to access both Extensional and Intensional Knowledge, database wide.

5. Prototype Implementation and Usage

There are two basic approaches to process intensional knowledge. They concern the location where the inference engine is placed, in terms of cooperation with the database and the user.

These are:

1) Tight Coupling, the inference engine is integrated with the database server.
2) Loose Coupling, a transparent interface between the user and the database.

The tight coupling integrates the inference engine very closely with the database. When the user issues a query, it is processed by the database, and if necessary, the database launches the inference engine. After then, the database generates the reply and sends it back to the user. Such an approach seems to be very effective in terms of performance. However, it narrows applicability of the technology to a particular RDBMS. Since one of the goals is to create a versatile system which may work with any RDBMS this approach is rejected (some further research regarding performance and the tight integration is conducted, though).

For the loose coupling, the inference system is placed, logically, between the user and the database system (see Fig. 3). The user issues a query, which is intercepted by the inference system. Then, the system checks if the query refers to any *Jelly View*s. If there is such a reference, then it downloads the programs for particular *Jelly Views*, and it starts the inference engine. When the inferred data is ready, the original query is sent to the database, in order to generate the reply. The database reply is based on both the database relation states and the inferred data. The reply is returned to the user.

If there is no reference to a *Jelly View*, then the query is forwarded to the database directly. The inference system works transparently then: if there is no need for the inference the query and the reply are passed without any modification.

This solution is flexible and applicable to any database system, as long as the communication method between the user and the inference system is the same as the communication method between the inference system and the database. The SQL queries and replies are passed between the user and the RDBMS through a communication interface (usually network transparent) then. The interface could be just plain text communication over the computer network or such a common protocol as the ODBC: Open Database Connectivity[4].

The inference system becomes a middleware between the user and the database. Having the network nature of the communication, the inference system may run on a machine different than the database system does. Actually, there could be even a farm of the inference systems, working as the middleware between users and the database. The inference systems may be distributed among many physical machines.

There is a significant overhead concerning the connection between the inference system and the database. In the loose coupling approach all data, including information about External Matching, Internal Matching, Logic Program, and inferred data as well, have to travel between two systems: the database and the inference engine. It becomes a performance bottleneck. There is a trade off, then: flexibility and versatility versus the communication bottleneck.

[4]Its specification is developed by Microsoft. Ironically, *Open* is just an empty phrase here, because ODBC remains tied up to Microsoft and it is not open at all.

There are the following features of the loose coupling:

- The system is hardware and software independent: applicable to virtually any database.
- It is scalable, supporting processing power distribution, the inference engine is allowed to run on a machine different than the database system.
- No modification to the database is necessary (except for creating appropriate relations for the Matchings and the Program).
- Such a middleware may be applied to an existing database system, without spoiling its current functionality.

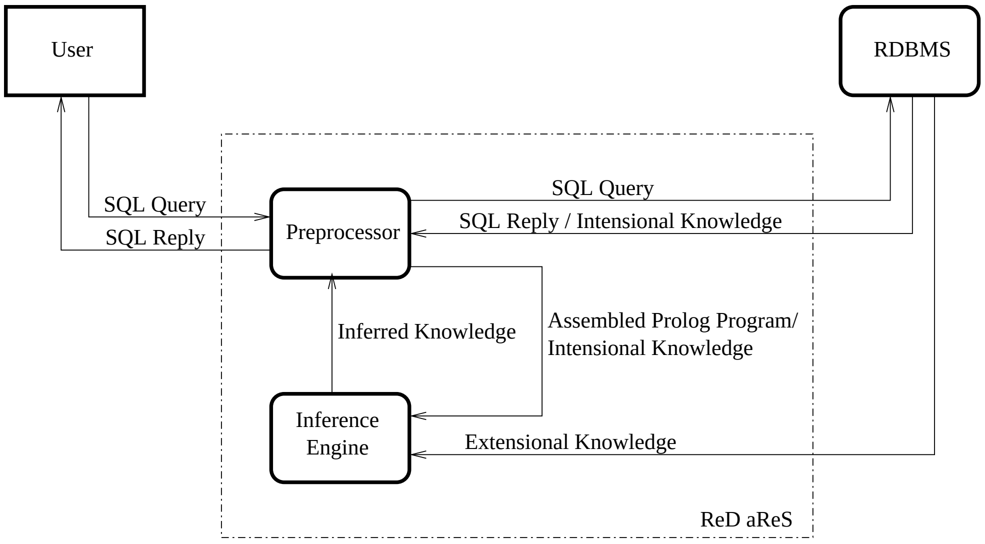


Fig. 3. Details on the Loose Coupling Architecture

The inference system is named *ReDaReS*, which is an acronym for: ***Re**lational **Da**tabase **R**ule **S**ystem.*

When the user issues a query, it is intercepted by ReDaReS. The system analyzes the query and confronts it with the External Matching, which is obtained from the database. It checks whether the query refers to a *Jelly View* or not. If the query does not refer to any *Jelly View*, then it is forwarded to the database without any changes. The reply from the database goes to ReDaReS, and then it is forwarded to the user. If the query refers to *Jelly Views*, then the Internal Matching and the Program for particular *Jelly Views* are brought from the database. Then, the PROLOG program is formed, and the inference engine is started.

The inference engine can access extensional knowledge (database relations) on demand, during run-time. The inferred data is generated and sent to the database as temporary relations. The user query is rewritten to address the temporary relations instead of the original *Jelly Views*. The rewritten query is sent to the database. The reply from the database is forwarded to the user.

The user-ReDaReS and ReDaReS-database communication method is chosen to be ODBC. It gives flexibility, and makes the inference system database independent, since most of RDBMS support ODBC. If the user query contains *Jelly View* names, it is rewritten in such a way that *Jelly Views* are replaced with temporary relations. Just for simplicity these relations are referred to as *Jelly Relations*. They are subsequently used to hold inferred data. The rewritten query is just a plain SQL query. The Jelly Relations are not in the database prior to issuing the user query.

After the query is rewritten, the program is assembled and the inference engine generates results which are sent to the database providing states of the Jelly Relations. The transmission overhead should be considered as the price which has to be paid to satisfy the requirement of flexibility of the system, that is to make it applicable to virtually any database.

To refer to a *Jelly View*, a PSM-like[5] syntax is used. Each *Jelly View* has its schema, which is provided by the External Matching (`tp-argument` and `table-predicate` entities in Fig. 2). For example, assuming that there is the following *Jelly View* schema defined: `jelly(first,second)`, this *Jelly View* can be queried as:

```
SELECT *
FROM jelly(,'SELECT 2') AS j1,
     jelly(,'SELECT 3') AS j2,
     other_relation
WHERE j1.first > 4,
      j2.first < 5
;
```

The *Jelly View* arguments are calculated from subqueries. In the example above, they are just simple `SELECT` queries, but the user may specify a subquery as complex as he or she needs. The result of such a subquery has to be a single column. So, it may pass not only a single value to the goal, but also a set of values, which are used as goal boudns by the inference engine. It makes the system even more flexible. For example, having the following query:

```
SELECT *
FROM jelly(,'SELECT id FROM other_relation')
;
```

the second parameter for the goal of the *Jelly View* `jelly` is taken as a set of all values of `id` attribute of relation `other_relation`. In other words, the state of `jelly` may contain, in the second column, `id` values taken from `other_relation` only. Values in the first column are unbounded and they will be inferred.

[5]PSM – Permanent Stored Module, also known as Stored SQL Function/Procedure

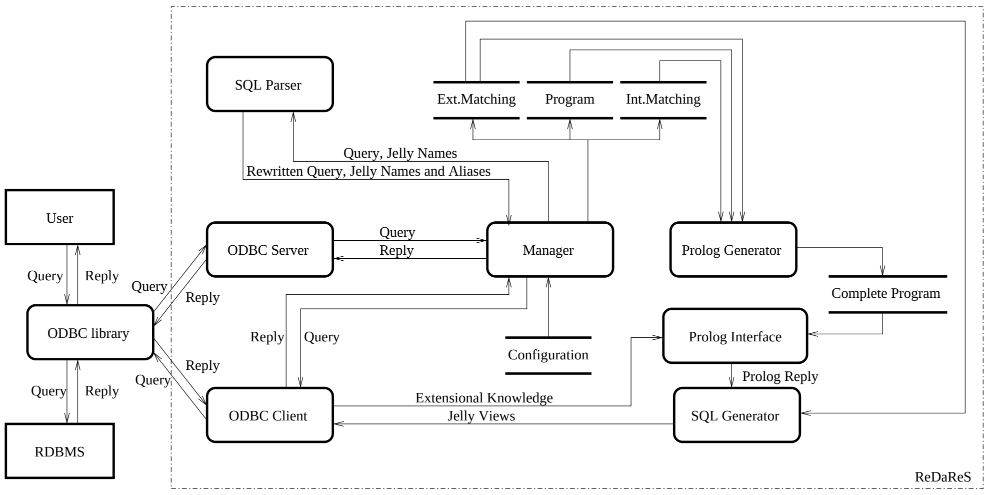


Fig. 4. ReDaReS, the Data Flow Diagram

The data flow diagram of a prototype system is given in Figure 4. The main component of the system is the Manager. It controls all other modules. If there is an incoming query it is received by the ODBC Server. Then it is sent to the Manager for further processing. The Manager consults the database concerning registered *Jelly Views* and sends the query to the SQL Parser along with this data. The SQL Parser processes the query and identifies *Jelly Views*. Then, the query is rewritten and information of all the *Jelly Views* in the query is returned to the Manager. If the query does not contain any references to *Jelly Views*, then it is sent to the ODBC Client, which connects to the database and executes the query. The results are returned to the Manager and forwarded to the user through the ODBC Server.

If the SQL Parser notifies that there are some references to *Jelly Views*, then the course of action is different. The Manager downloads information about the External Matching, Internal Matching and the Program concerning each *Jelly View*, and starts the Prolog Generator. The Prolog Generator uses the Matchings and the Program to assemble a Complete Program, which is capable of satisfying the goals defined by the External Matching.

Next, the Prolog Interface is launched. It uses the Prolog inference engine to satisfy the goals and it is capable of downloading extensional knowledge from the database. The Prolog Interface provides Prolog Reply which is the inferred data. Then, the SQL Generator takes over. It feeds, using the External Matching, the database with the *Jelly Views* states. More precisely, these are states of the relations which are replacing references to the *Jelly Views*, in the rewritten query – the Jelly Relations. As a result the database possesses the inferred data. Finally, the Manager forwards the rewritten query to the database through the ODBC Client. The database sends back a reply, which is forwarded to the user through the ODBC Server.

6. Performance and Conclusions

A series of experiments have been carried out to investigate ReDaReS performance. The main test subject is the tree traversal problem. It is finding predecessor or ancestor nodes in a tree structure stored in the database as a relation.

The test tree is composed of 12 levels, having 3 children at each node. The number of nodes is given as a Sum of Geometric Series: $S = a + ar + ar^2 + \ldots + ar^{n-1} = a\frac{1-r^n}{1-r}$, and with $a = 1, n = 12, r = 3$, there are: $S = 265720$ nodes. Such a tree structure is represented as a single relation: `subject(Parent_id, Item_id, Name)`. Where `Parent_id` is a numerical identifier of the parent node (foreign key), `Item_id` is a numerical identifier of the node, and `Name` is the node's name.

The PROLOG code for finding relationships among nodes is given below:

```
find(Parent,Child):- tree(Parent,Child,_).
find(Parent,Child):- tree(Parent,C1,_),
                     find(C1,Child).
```

In order to generate a *Jelly View*, which is capable of finding the relationships, the above program is decomposed. The *External Matching* is set to define the *Jelly View* which has the following schema: `find(parent_id, child_id)`, and uses `find/2` as the goal. Furthermore, the *Jelly View* corresponds to the above clauses. The *Internal Matching* defines, that simple clauses (facts) of the `tree/3` predicate are taken from the `subject` relation.

There has been a series of experiments carried out in an isolate environment (with the PostgreSQL RDBMS, and ReDaReS running only[6]). In general, they focused on finding all child nodes of the node at different levels of the tree structure. In particular these levels are: 11, 9, 7, 5, 3, 1, where the level number 1 is the root. The query, used in the experiments, finds all child nodes of the given parent one. The parent node is selected by passing the first argument to the *Jelly View* and setting the second one unbounded. The precise value passed as the `parent_id` is a result of a sub-query given as the argument. The query is given below.

```
SELECT * FROM find('SELECT XXX',);
```

where `XXX` is the following `parent_id`s queried in turns: 29523, 3279, 363, 39, 3, 0, which correspond to the levels number: 11, 9, 7, 5, 3, and 1 of the tree structure. The chart in Figure 5 shows the results.

The X-axis is the number of nodes obtained from the query (its different for different levels of the tree). The Y-axis is the elapsed processing time which includes the RDBMS processing time, the ReDaReS processing time and the communication overhead.

The performance of ReDaReS is compared with the performance of a PSM with the same functionality. The PSM is written in PL/pgSQL, which is a native PostgreSQL procedural language.

[6]The computer system was besed on Mobile Intel Celeron CPU 1.50GHz, using Linux 2.4.21, PostgreSQL 7.2.1, Unix ODBC 2.1.1, and SWI-Prolog 5.0.10

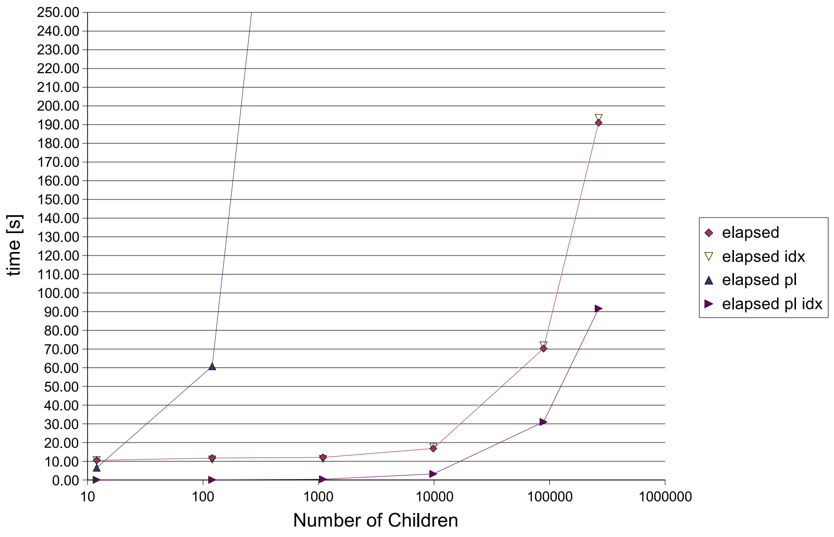


Fig. 5. Performance of the System – Browsing a Tree

The experiments take into account the database indexing as well. The graphs labeled elapsed and elapsed idx represent the timings of ReDaReS on the relation `subject`, without and with the indexing turned on respectively. The graphs elapsed pl and elapsed pl idx represent the timings provided by the PSM without and with the indexing. As it is showed, the ReDaReS system outperforms the PSM if there is no indexing involved. Turning the indexing on (on the attributes of the `subject` relation) has a tremendous impact on PSM based solution, which becomes faster than ReDaReS.

Some of the ReDaReS slowdowns regard the fact, that the system has to feed the database with the results from the inference process. The second set of experiments takes it into account (see Fig. 6). This time the output tuples are discarded. They are not generated by ReDaReS nor PSM. Such an approach investigates timings of the inference process alone, without the communication overhead (which is pretty significant considering 265720 tuples returned while querying at level 1).

The *entry time-delay* of the ReDaReS system is caused by the necessity of downloading extensional knowledge into the inference engine, plus the time needed to start the engine up. It takes about 10 seconds for this particular experiment, which is downloading 265720 tuples into the inference engine.

The experiments show that ReDaReS is more efficient than the PSM approach concerning the inference process itself. The conclusion is that the system is sufficiently efficient. However, its performance can be improved by optimizing the communication between the prototype system and the database.

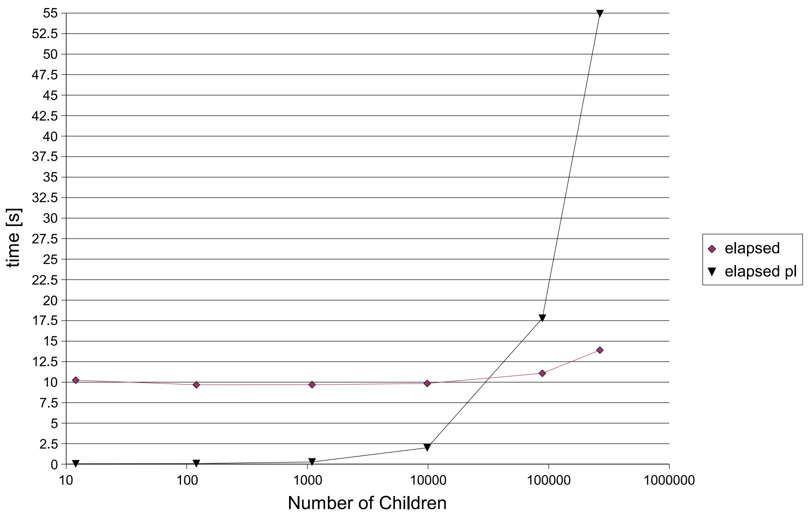


Fig. 6. Performance of the System – Browsing a Tree (Output Discarded)

The improvement should focus on both downloading extensional knowledge and uploading the results of the inference process.

Summarizing, the *Jelly View* technology extends the Relational Database systems, in such a way, that even more complex problems than these specified in Section 1 can be smoothly approached keeping SQL as outermost communication technology. These problems are tackled by introducing rule-based processing to the database systems. The functionality of Relational Databases is significantly extended towards that of Deductive Databases [4, 9, 12], by integrating the proposed technology into a database. The technology is based on coupling the existing PROLOG inference engine with the database.

The prototype ReDaReS system, providing the proposed technology, has been implemented and tested on a number of problems [14]. It has shown out the following key properties. The original functionality of the database is preserved. Both data and knowledge are stored within the Relational Database; no additional knowledge base is necessary. There are modules specified in PROLOG for extending data processing capabilities, which are decomposed and stored as data in the database. Results of the inference process, which is inferred knowledge, are accessible as dynamically generated SQL views; the necessary code is generated on request from components stored in the database. The communication method between the user and the database remains SQL. Above properties meet the requirements stated in Section 3.

Obviously, the tree traversal problem (see Figure 5, 6) is of exponential computational complexity. However, the technology and its implementation turned out to be relatively efficient, so that even practical, realistic problems can be solved with this simple approach. Taking into account that at present no further optimization of the code nor other mechanisms (such as use of heuristics or constraints) have been considered and the experiments were carried out with PC class computers, the technology seems to be a promising extension to classical RDBMS.

The proposed solution is more flexible than PSMs. It allows modular programming, and may be supported with a CAD system easily. Furthermore, it is designed to work with any database system regardless of its PSM capabilities. The performance of the prototype system is at least comparable with this of server-based processing (PSM). There is also an ongoing research in this domain, which is focused on the decomposition of intensional knowledge and improvement of the system efficiency.

References

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

[2] Connolly T., Begg C., Strachan A.: *Database Systems, A practical approach to Design.* 2nd ed., Implementation, and Management. Addison-Wesley, 1999.

[3] Covington M. A., Nute D., Vellino A.: *Prolog programming in depth.* Prentice-Hall, 1997.

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

[5] Elmasri R., Navathe S. B.: *Fundamentals of Database Systems.* Addison Wesley, 2000.

[6] Garcia-Molina H., Ullman J. D., Widom J.: *Database Systems, the complete book.* Prentice Hall, 2002.

[7] *Srini Venigalla Netsetgo. Expanding recursive opportunities with sql udfs in db2 v 7.2.* Technical report, International Business Machines Corporation, 2002.

[8] Nilsson U., Małuszynski J.: *Logic, Programming and Prolog.* John Wiley & Sons, 1990.

[9] Ramakrishnan R., Srivastava D., Sudarshan S., Seshadri P.: *The CORAL deductive system.* VLDB Journal: Very Large Data Bases, 3 (2), 1994, p. 161–210

[10] Ullman J. D.: *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.

[11] Ullman J. D., Widom J.: *A first course in Database systems.* Prentice-Hall, 1997.

[12] Vaghani J., Ramamohanarao K., Kemp D. B., Somogyi Z., Stuckey P. J., Leask T. S., Harland J.: *The aditi deductive database system.* Technical report, University of Melbourne, 1994.

[13] Vlahavas I., Bassiliades N.: *Parallel Object-Oriented, and Active Knowledge Base Systems.* Kluwer Academic Publishers, 1998.

[14] Wojnicki I.: *A Rule-based Inference Engine Extending Knowledge Processing Capabilities of Relational Database Management Systems.* (Ph.D. Thesis), AGH University of Science and Technology, 2004. A copy is availbale upon request from the author: `wojnicki@agh.edu.pl`

[15] Wojnicki I., Janikow C. Z.: *Extending data processing capabilities of relational database management systems.* In Arabnia H. R., Joshua R., Mun Y. (Eds), International Conference on Artificial Intelligence, v. I, p. 388-393, Las Vegas, Nevada, USA, CSREA Press, 2003

[16] Wojnicki I., Ligeza A.: *An inference engine for rdbms.* In 6th International Conference on Soft Computing and Distributed Processing, Rzeszów, Poland, 2002