VLDB Journal, 3, 211-243 (1994), Kotagiri Ramamohanarao, Editor
©VLDB

DECLARE and SDS: Early Efforts to Commercialize Deductive Database Technology

Werner Kießling, Helmut Schmidt, Werner Strauß, and Gerhard Dünzinger

Received April, 1993; revised version accepted December, 1993.

Abstract. The Smart Data System (SDS) and its declarative query language, Declarative Reasoning, represent the first large-scale effort to commercialize deductive database technology. SDS offers the functionality of deductive reasoning in a distributed, heterogeneous database environment. In this article we discuss several interesting aspects of the query compilation and optimization process. The emphasis is on the query execution plan data structure and its transformations by the optimizing rule compiler. Through detailed case studies we demonstrate that efficient and very compact runtime code can be generated. We also discuss our experiences gained from a large pilot application (the MVV-expert) and report on several issues of practical interest in engineering such a complex system, including the migration from Lisp to C. We argue that heuristic knowledge and control should be made an integral part of deductive databases.

Key Words. Declarative reasoning, query optimizer, multi-databases, distributed query processing, heuristic control, productization.

1. Introduction

Despite a high volume of deductive database research (Gallaire et al., 1978, 1984; Bayer, 1985; Ullman, 1985; Zaniolo, 1985; Tsur and Zaniolo, 1986; see also Ceri et al., 1989; Naqvi and Tsur, 1989; Ullman, 1989), only a handful of deductive database systems have been prototyped, and very few have been used for realistic

Werner Kießling, Ph.D., is Professor of Computer Science, Lehrstuhl für Informatik 2, Universtät Augsburg, Universistätsstraße 8, D-86135 Augsburg; Helmut Schmidt, Ph.D., is Manager Software Developer, Süddt.Finanzsoftware GmbH, Reifersbrunner Str. 26, D-86415 Mering; Werner Strauß, M.S., is Software Consultant, PECOM Unternehmensberatung GmbH, Otkerstr. 7b, D-81547 München; Gerhard Dünzinger, M.S., is Senior Software Engineer, ABLAY Optimierung GmbH, Schmidgern 1, D-82205 Gilching.

applications. Among them are LDL (Tsur and Zaniolo, 1986); NAIL! (Morris et al., 1986) and its successor Glue-NAIL! (Phipps et al., 1991); EKS-V1 (Vieille et al., 1990); Starburst (Mumick et al., 1990); Aditi (Vaghani et al., 1991); LOLA (Freitag et al., 1991); and CORAL (Ramakrishnan et al., 1992).

In this article we focus on the development of the Declarative Reasoning language (DECLARE), the Smart Data System (SDS), and the commercialization of deductive database technology at a very early stage. Except for private communications and colloquium talks, the only available documentation was internal reports (e.g., Kießling, 1987) and a brief overview (Kießling and Güntzer, 1990). Our initial experiences with this new field were at the Technical University of Munich. In 1986, a large research and development center was built under the direction of the first author to turn these new ideas into a commercial product.

For a better understanding of some design decisions within this project, we offer some remarks on the marketing strategy. One main selling point of this technology is its strategic decision making capability. Database technology, enhanced by deductive rule-based capabilities, can assist enormously in condensing information to make good decisions (a major key in achieving a competitive advantage). In many corporate decisions, relevant information is spread out over heterogeneous databases, and such an environment has to be addressed.

The design and development of DECLARE and SDS (from 1986 to 1990) were particularly challenging and exciting—many of the beneficial results known today (e.g., model-theoretic, optimization, or interoperability issues for multi-databases) were barely emerging while the development, with all its milestones, was in progress. This article gives an insight into our efforts.

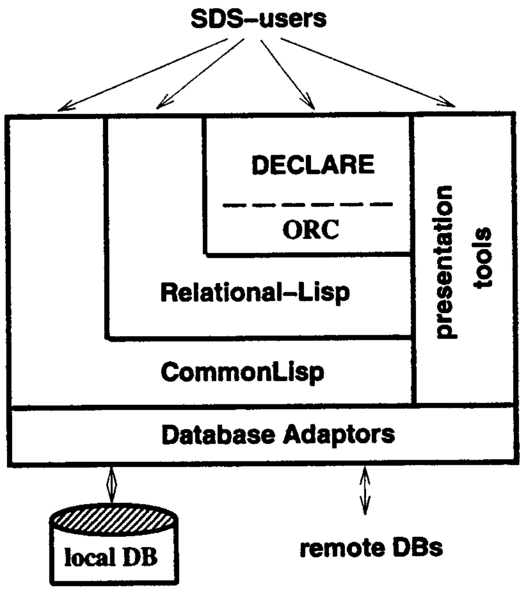
The rest of this article is organized as follows: In Section 2 we introduce the architecture of SDS and discuss deductive reasoning in a heterogeneous database environment. In Section 3 we describe our site-transparent, declarative query language, DECLARE. We discuss various aspects of Relational-Lisp, the runtime environment for the Lisp-version of DECLARE, in Section 4. We dedicate Section 5 to the crucial issue of query optimization, and give some detailed case studies. In Section 6 we report on a successful pilot application of this technology, the MVV-Expert. Section 7 is concerned with aspects of productization and commercialization. In Section 8 we tackle one weak point of the puristic declarative view and summarize some of our experimental and theoretical results on entering heuristic user knowledge and control. In Section 9 this article ends with some of the lessons we have learned.

2. Architecture of SDS

2.1 Heterogeneous Database Reasoning

Novel programming paradigms with the complexity of deductive databases can be marketed effectively only if past user investments in database technology are preserved to a high extent (notably more recent investments in SQL databases).

Figure 1. Architecture of SDS



SDS is a deductive database system that was designed to meet these requirements. Although it can also operate in a stand-alone mode, a main objective of SDS is to add powerful deductive capabilities and connectivity to existing relational databases. Therefore it is not required to store both the extensional part (i.e., ground facts) and the intensional part (i.e., rules that are not ground facts) of a deductive database. Instead, the extensional part can be stored in different relational database systems, and be accessed dynamically via so-called database adaptors. The advantages of deductive reasoning thus can be combined with the strengths of relational databases to develop and maintain large-scale applications.

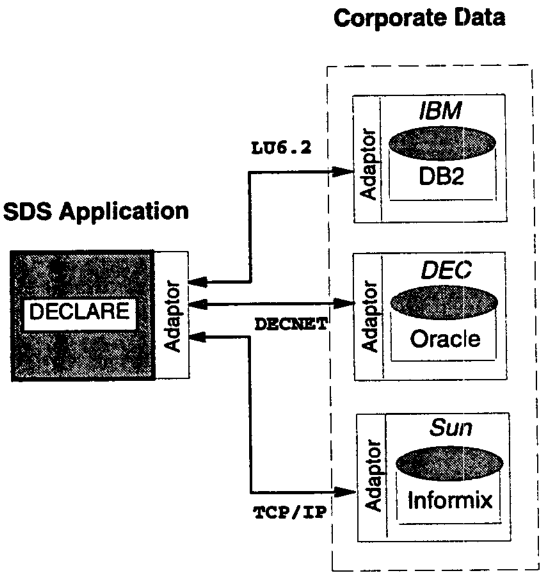
Let us explain this architecture for heterogeneous database reasoning in more detail. SDS consists of the following major components:

- The declarative query language DECLARE.
- The optimizing rule compiler (ORC).
- The extented relational algebra (ERA).
- Various database adaptors to commercial SQL-systems.

ERA is the runtime environment for compiled and optimized DECLARE queries. During the project two delivery platforms emerged, one in a Lisp-environment (mainly used for internal application development), the other as a C-based production environment (Section 7). Figure 1 shows the architecture of SDS, with Relational-Lisp as the implementation of the ERA.

Relational-Lisp (Section 4) adds relations as first class objects to the Lisp environment. The local database may be loaded into virtual Lisp memory or be

Figure 2. Database adaptors for heterogeneous reasoning support



disk-resident, managed by an upgraded high-speed SQL database kernel. Relational-Lisp also interacts with heterogeneous databases through the database adaptors. An integrated architecture is realized for the local database, whereas tight coupling is implemented for remote databases.

2.2 Database Adaptors

In 1986, the area of interoperability for multi-databases was a new research topic and off-the-shelf database connectivity tools were not available. We therefore had to develop our own database adaptors (despite the development of the SQL/1 standard), which also had to deal with interoperability in a heterogeneous SQL-environment. Figure 2 shows a sample SDS configuration with some actually realized adaptors. Note that during an application session several remote databases (in addition to the local one) can be accessed within a single DECLARE query.

The database adaptors in SDS are in charge of the following tasks:

- Given some transport protocol, establish a connection to a relational DBMS.
- Get catalog information about the relations stored in the relational DBMS. This information is needed by the ORC to check and optimize a DECLARE program.
- Translate an ERA-expression into the SQL dialect of the target relational DBMS.
- Transmit this query to the relational DBMS and start query execution.
- Transmit the result of this query back to the ERA and perform type transformations (due to differences in the type systems).

3. The DECLARE Language

From a user's point of view, the most important part of a deductive database system is its declarative rule language, which offers expressiveness and convenience far beyond that of a conventional SQL database system. The DECLARE language is a dialect of the *Datalog*-family. Besides defining virtual relations by rules, DECLARE is used to specify queries to a deductive database. Set-oriented updates to the base relations are done using Relational-Lisp (or Embedded SQL for the C-version).

3.1 DECLARE Features

The DECLARE language offers the following features:

- Full recursion (Section 5.6).
- Negation (Section 5.6).
- All-quantification in rule bodies (Example 1 and Section 5.6).
- Interpreted function terms, written in a host language (CommonLisp or C). Arithmetics, set and list processing, and abstract data types can be realized through this mechanism.
- Constraints with predefined and user-defined boolean predicates.
- Grouping and aggregation (Example 2).
- Site-transparent access to ground facts stored in local or remote databases.

The DECLARE language is *strongly typed*. All attribute values and variables must belong to one of the DECLARE data types. Besides the basic types *integer, float,* and *string*, *decimal, date, time,* and *timestamp* also are offered (this is the SQL/2 standard). Moreover, to support complex objects the data type *list* is provided. For ground facts stored in external databases an automatic mapping from the data types of the specific DBMS to the corresponding DECLARE data types is performed. This type system comes with a variety of conversion and extraction functions to translate between representations.

Restrictions of the DECLARE language are:

- Range-restricted variables.
- Covered variables in negation and all-quantification.
- Stratified negation and grouping.

The declarative semantics of DECLARE is the usual perfect model semantics with an equivalent bottom-up fixpoint evaluation procedure (van Emden and Kowalski, 1976; Apt et al., 1987; Przymusinski, 1987). DECLARE is completely order-independent with respect to predicates in rule bodies, implying in particular that left-recursion causes no safety problems as in Prolog.

If declarative languages are to be incorporated into conventional database processing, then the addition of features for grouping and aggregation is indispensable (compare group-by-having in SQL). From early exposure to customer needs it became apparent that aggregation must be supported within recursion, leaving the sheltered paradise of stratified logic programs. Therefore, the stratification requirement was enforced only for the "normal" use. Beyond that, DECLARE supports a default operational semantics for non-stratified recursion (Section 5.5).

DECLARE integrates rule-based programming with the full functionality of applicative programming, because DECLARE programs are embedded in CommonLisp or C. On the other hand, CommonLisp or C functions may be used within DECLARE rules. A benefit of this flexibility arises when an attribute is regarded as an abstract data type. By choosing an appropriate implementation, the efficiency of DECLARE programs may be enhanced (see e.g., the N-queens benchmark in Kießling, 1992).

3.2 DECLARE Examples

Now we give two brief examples of DECLARE. In contrast to the usual Horn clause Prolog-style, rules with the same head predicate are collected and define a *virtual relation*.

Example 1: (Use of the forall-construct, of constraints, and of arithmetics.)

If in 1989 the total sales of a sales person were $1,000,000 or more, and if all her/his customers were highly satisfied in 1989, then she/he shall get a special bonus of 0.5% of the sales exceeding $1,000,000.

```
DEFINE VIRTUAL RELATION special_bonus(sp string,bonus decimal) {
  IF  total_sales(sp, 1989, sum) WITH sum >= 1000000,
        FORALL cust
            (customer(sp, cust), sales(cust, _ , 1989) IMPLIES
             cust_satisfaction(cust, 1989, sat)
             WITH sat IN ("high", "very high"))
  THEN special_bonus(sp, 0.005 * (sum - 1000000)); }
```

Get all sales persons whose special bonus is more than $50,000.

```
DEDUCE (sp) FROM special_bonus (sp, bonus) WITH bonus > 50000;
```
□

Example 2: (Use of grouping with aggregation.)
Determine the total sales of each sales person per year.

```
DEFINE VIRTUAL RELATION
  total_sales (sp string, year integer, sum integer) {
  GROUP customer (sp, cust), sales(cust, value, year)
  BY    sp, year
  TO    total_sales(sp, year, SUM(value)); }
```
□

4. Relational-Lisp

In this section we present the extended relational algebra (ERA), which is the target language of the optimzer. We focus on Relational-Lisp,[1] implementing the ERA for the Lisp-version of DECLARE (the C-version is dealt with in Section 7).

4.1 Extended Relational Algebra

Relational-Lisp, the runtime environment of DECLARE, offers the following ERA operations:

```
project, select, product, equijoin, thetajoin, semijoin,
antijoin, difference, intersect, union, groupby, iterate
```

Samples of ERA code are provided by the case studies in Section 5.6. The main extensions to the standard relational algebra are as follows:

- The semi-naive operator `iterate` for efficient least fixpoint computations of general recursive predicates implements the optimizations proposed by Güntzer et al. (1987). Formal differentiation of the fixpoint equation is done at the level of ERA expressions (Bancilhon, 1986; Schmidt, 1986).
- Interpreted host language functions may occur in `select, thetajoin, project`.
- `Antijoin` is added to implement negation.
- `Groupby` is a complex grouping operation with aggregation.
- Pipelining of select-project; and join-select-project; sequences is provided.

It should be noted that versions of the standard semi-naive operator have been invented independently at various places (e.g., Balbin and Ramamohanarao, 1987). Sophisticated refinements of `iterate` (Section 8) seem to be a speciality of DECLARE, however.

Several *runtime optimizations* are performed by these operations:

- Index selection.
- Creation of temporary indexes (e.g. in hash-based joins).
- Simple algebraic optimizations (e.g., $R \bowtie \emptyset = \emptyset$).

1. A first operational version of Relational-Lisp was presented at the Hannover fair in the spring of 1987 and was fully released one year later (Hillman, 1988).

Additional functions are provided to scan relations. Thus result relations can be processed by the non-deductive modules of an application program (e.g., to generate graphic output). A complete abstract data type for working with relations is available, together with the data definition operations described below.

4.2 Global Schema and Transaction Management

Relational-Lisp supports different types of relations:

- *Temporary relations* are generated as the result of ERA-expressions, but can also be defined explicitly.
- *Permanent relations* may be local relations (i.e., maintained directly by Relational-Lisp), or external relations residing on remote SQL-databases. Primary indexes and any number of secondary indexes can be created if desired (`create_index`, `drop_index`).
- *Catalog relations* contain the usual meta-data plus information on accessible databases, their locations, and the necessary parameters for accessing them.

For updating relations, Relational-Lisp offers the usual functionality. *Transaction processing* is supported via *sessions* (`start_session`, `commit`, `rollback`, `end_session`). As a specialty, local transactions may even span data definition commands. Since most commercial SQL-databases did not support a prepared-to-commit state at that time, a distributed two-phase commit for multi-database updates could not be attempted.

During a session, external databases can be plugged in and de-plugged dynamically to restrict the allocation of resources to the phases when queries are evaluated and particular external relations need to be addressed. Relations can be *imported* from accessible databases. This means that schema information is extracted automatically and the relations become accessible under a logical name. The data, however, remain at the remote database and relevant pieces are fetched only at query evaluation time (tight coupling). The following code fragment demonstrates the access to remote relations:

```
(create_db_access :domain_name "MVV"  :db_name "timetable"
                  :external_name "/usr/db/mvv"
                  :host_name "server1"  :type "oracle"
                  :transport_protocol "tcp_ip"   ...)
(start_session "MVV")
(plugin_db "timetable")
(import_rel "departure_time")
(end_session :commit)
```

Now the relation `departure_time` is accessible in subsequent sessions when the database `timetable` is plugged in. It can be accessed in a site-transparent way like any other local or external relation.

In summary, Relational-Lisp acts as an *SQL-integrator* in heterogeneous multi-database environments with convenient global schema integration facilities. Since DECLARE programs are executed within Relational-Lisp sessions, all transactional features apply to such DECLARE applications automatically.

4.3 Distributed Query Evaluation

Operations that involve more than one external relation are executed in a distributed fashion whenever possible. Here reduction of network traffic is an important optimization objective. The class of feasible algorithms, however, is constrained because, for query purposes, external database administrators normally don't admit write operations (e.g., to temporarily store intermediate query results). Under these premises a simple equijoin $R \bowtie_{r=s} S$ involving two relations on different external databases can be performed in the following way:

1. Determine the smaller relation R from catalog information and compute a join filter for attribute r:
   `filterR := db_adaptor("select distinct r from R");`
2. Get all join partners for attribute s:
   `tmpS := db_adaptor("select * from S where s in :filterR");`
3. Compute a join filter for attribute s and get all join partners for attribute r:
   `filterS := project(tmpS, s);`
   `tmpR := db_adaptor("select * from R where r in :filterS");`
4. Construct the join result locally:
   `result := equijoin(tmpR, r, s, tmpS);`

Note that when the filters are transported via the in-clause of Dynamic SQL, the vendor-dependent limits on the length of an SQL statement must be observed. In such cases, the SQL query is split automatically into two or more queries, each containing some portion of the filter.

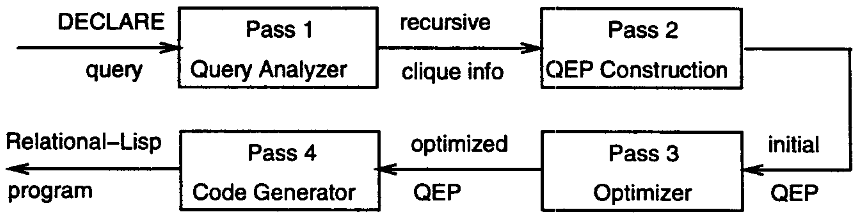
5. The Optimizing Rule Compiler (ORC)

5.1 Phases and Passes of the ORC

DECLARE is a static compile-time language for defining virtual relations via rules, for asking ad-hoc queries, and for defining query forms for repetitive use. Since different tasks have to be done for rules and queries, the ORC could easily be divided into two main phases: a *rule processing phase* and a *query processing phase*.

The further subdivision of these phases into passes and, in particular, the definition of the internal data structures required some crucial design decisions affecting the modularity, flexibility, and efficiency of the ORC. Our guiding policy for choosing and implementing internal data structures was to use tailored data structures where needed, and to stick to relations whenever possible. The latter policy,

Figure 3. Query processing: Main passes of the ORC



of course, reduced specification and implementation time, because DECLARE and Relational-Lisp could be used to implement some parts of the ORC.

Rule processing. During rule processing the DECLARE parser performs the usual tasks; the rules are syntactically checked and transformed into an internal representation. After all rules are parsed, context-sensitive checks like the stratification test are done. The chosen internal representation is a set of relations and provides a one-to-one mapping of the source rules to their internal format. Additionally, a predicate connection graph (PCG) is maintained as a relation. Recursive cliques (also called strongly connected components) are computed easily using this PCG.

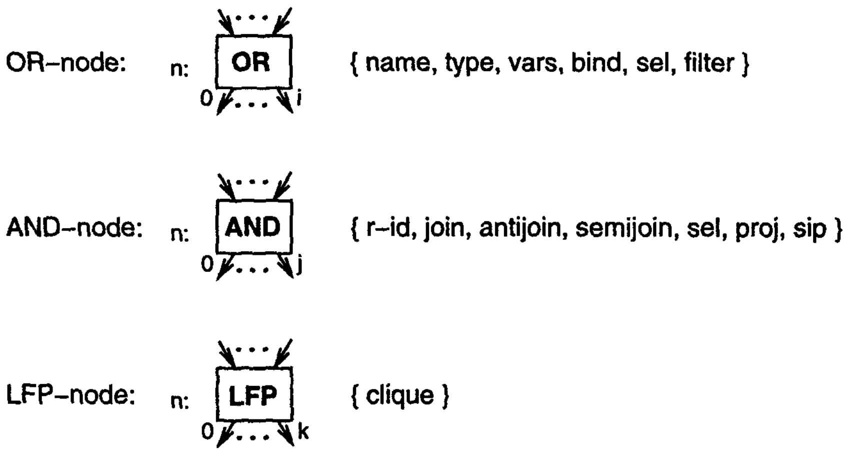
Query processing. Query processing to answer an ad-hoc query or to compile a query form is divided into four main passes (Figure 3). Pass 1 syntactically checks the query. Pass 2 constructs the initial query execution plan (QEP). The QEP is the main data structure of the ORC and will be discussed later. The initial QEP is handed over to pass 3, which handles all optimizations. Optimizations are performed exclusively by QEP transformations (Section 5.4). The result of pass 3 is an optimized QEP, which is the input for pass 4, the code generator. Here an ERA-expression is generated in the form of a Relational-Lisp program (in the C version of DECLARE Relational-C code is produced here; Section 7). This Relational-Lisp program either can be executed immediately for ad-hoc queries or be stored permanently for later recurrent use in case of query forms.

5.2 QEP Data Structure

The QEP is the central data structure of the ORC and has to represent the *operational semantics* of a DECLARE query. The following rationale guided the choice for the QEP data structure.

- The class of QEPs has to be powerful enough that a broad variety of optimization methods can be described as transformation schemes between

Figure 4. QEP-nodes and their adornments


OR–node:
n:
OR
0 ... i
{ name, type, vars, bind, sel, filter }
AND–node:
n:
AND
0 ... j
{ r–id, join, antijoin, semijoin, sel, proj, sip }
LFP–node:
n:
LFP
0 ... k
{ clique }

QEPs. Moreover, extensibility is of utmost concern since new optimization methods may be invented any day.

- The modeling of the query answering process through the QEP must be both abstract enough for optimization purposes (otherwise certain transformations may become awkward or even impossible) and concrete enough for code generation (which should be straightforward and compact).

These considerations have led to the following definition of the QEP as a special query/rule-graph (Ullman, 1985), which has a direct interpretation as an ERA operator graph.

Definition: (*The QEP as adorned "dag"*)

- A QEP Q = (N, E, A, ν, α) is a finite, ordered, directed, acyclic, and single-rooted graph with nodes N, edges E $\subseteq$ N $\times$ N, a set A of adornments and two labeling functions ν : N $\longrightarrow$ $\{$OR, AND, LFP$\}$ and α : N $\longrightarrow$ A.

- A QEP is bi-partite with nodes partitioned into $\{$AND, LFP$\}$-nodes and OR-nodes. The root is an OR-node. ☐

Note that we have defined the class of QEPs as "dags" (instead of trees) to support *common subexpressions*. The ordering imposed on son nodes is used to represent join orderings for sideways information passing. QEP nodes are graphically represented as shown in Figure 4.

The adornments have the following meaning:

- For OR-nodes, `name` is a relation name (virtual or base), `type` distinguishes between different types of relations, `vars` is a list of variables relevant at this node, `bind` contains binding information (`b` if bound, `f` if free) for the variables in `vars`, `sel` describes selection conditions and `filter` is a set of variables for which dynamic filters are applicable.

- For AND-nodes, `r-id` is a system-generated rule identifier, `join` describes "genuine" Θ-join conditions whereas `antijoin` (`semijoin`) holds antijoin (semijoin) conditions, `sel` describes selection conditions, `proj` describes pipelined projections (applicable after join/antijoin and selection), and finally `sip` contains information for dynamic filter propagation (sideways information passing). Antijoin conditions correspond to negation and universal quantification. For rules with grouping and aggregation, `proj` also describes the aggregations to be performed.

- For LFP-nodes we have the `clique` adornment which lists the predicates in a recursive clique (mutual recursion is supported).

This set of adornments must be extensible according to the ORC-builders' needs. For example, one can add more information on special recursion types (e.g., `monotonic`, `transitive-closure`, or `bounded`, etc.) to LFP-nodes as soon as the optimizer can capture such cases by specialized algorithms.

The construction of the initial QEP follows an obvious procedure. For examples refer to the case studies in Section 5.6 and the QEPs displayed in the Appendix.

5.3 The Optimization Model

The task of the ORC is to successively transform the initial QEP to reduce the runtime costs for answering the given query as much as possible. Due to the complexity of the optimization problem, however, it will often be impossible to find the optimal QEP. Therefore, the only goal of the optimizer is to find a sufficiently efficient QEP.

All optimizations applied by the ORC have been implemented as *equivalence-preserving QEP transformations*. More precisely, the ORC provides a repertoire of modular and independent optimization methods with the following properties:

- Each transformation is applied to a part of the QEP (usually a node and all of its direct successors).
- No implicit assumptions are made (e.g., about the order in which the transformations are applied). It first tests whether all of its *pre-conditions* hold.
- Then, using *statistical* knowledge or some *heuristics,* it transforms the part of the QEP on which it is working.
- A transformation may change adornments of existing nodes, delete some nodes or sub-trees, or add new nodes to the QEP.

Besides the QEP transformations, a *search strategy*, describing when and where to use which transformation, is needed. For example, a possible search strategy for the optimization techniques `push-selection` and `push-projection` may be to perform two depth-first traversals of the QEP. During the first one, `push-selection` is applied to each node of the QEP, while during the second one, `push-projection` is applied to each node.

Note that the optimization process can be stopped at any time between transformations and code generation can be started. This feature is very useful if thresholds for elapsed optimizer time must not be exceeded.

This *modular approach* provides the following advantages:

- New optimization methods can easily be added, existing methods can easily be changed, different methods implementing the same optimization techniques can be compared.
- The order of applying transformations can be changed. For example, when adding a `common-subexpression` optimization, it is not obvious whether it is better to apply `common-subexpression` before `push-selection` and `push-projection` or the other way round.
- Several "simple" transformations can be combined into one "complex" transformation.
- Different search strategies and/or different opimization methods can be used for different areas of the QEP. For example, `join-order` optimization using statistical knowledge is done only for (sub-) QEPs with a height below a pre-definable threshold. For more complicated QEPs a heuristic join order optimization criterion based on bound/free information is used.

During the design and implementation of the ORC, many of the optimization techniques dealing with recursion were barely emerging. Therefore, this modular approach proved very effective. In early stages of the optimizer project (when the repertoire of optimization methods just contained some simple optimizations like `push-selection` and `push-projection`) the QEP was traversed several times and in each traversal one optimization was applied to all nodes. When more and complex optimizations were available, this simple strategy no longer produced satisfactory results. Using combined transformations we eventually got much better results.

5.4 Basic Optimization Repertoire

Choice of basic optimization methods:

```
push-selection, push-projection, join-order, specialize-operation,
push-filter, supplementary-magic-set, common-subexpression
```

Note that each optimization method can be switched on/off independently by annotations, and new methods can be added in a modular fashion as soon as they become available.

The modular, flexible, and integrated design of the ORC has been accomplished independently from other concurrent projects. Concerning the applied optimization methods, DECLARE borrows from other theoretical works, of course. `Push-selection` and `push-projection` are extended to push through recursion (Ramakrishnan et al., 1988). `Join-order` optimization determines a good ordering of the subclauses. Our heuristic algorithm for subgoal reordering is based on the *"bound-is-easier"* assumption (van Gelder, 1986; Ullman, 1989) and has polynomial complexity. `Specialize-operation` does things like substituting a `semijoin` for an `equijoin` immediately followed by a `project`, if possible. The `push-filter` transformation applies the ideas of *dynamic filters* (Kießling, 1986). The `supplementary-magic-set` implements the ideas of Bancilhon et al. (1986), Beeri and Ramakrishnan (1987), and Sacca and Zaniolo (1987).

Uncontrolled rewriting considered harmful. We decided to do the `supplementary-magic-set` transformation by a QEP transformation rather than by rewriting on the level of source rules, as usually suggested. Although this QEP transformation looks more complicated (Section 5.6), we believe that this approach provides some advantages over the usual rewriting. Separating rule rewriting for magic-set transformations from the rest of the optimization process may produce an overly lengthy and complicated program due to the "pump-up" factor inherent in the magic-set approach, which may lead to an explosion of the rules generated. In contrast, if magic-sets are integrated with all other optimization methods, a much more intelligent control of the transformation process becomes feasible. As an example, the ORC needs not apply the `supplementary-magic-set` transformation, if `push-selection` was already performed for the recursive clique in question. In this way the advantages of a modular optimizer design are carried over even for sophisticated recursive optimization methods.

5.5 Code Generation

Code generation is done by a *post-order* traversal of the QEP (i.e., the codes of the son nodes are generated before the code of the father is constructed). Note that, for stratified programs, such a post-order traversal respects the intended layer-by-layer evaluation. OR-nodes, AND-nodes and LFP-nodes are mapped to `union`, joins (`thetajoin, antijoin, semijoin`) and semi-naive iteration (`iterate`), respectively. For each type of node, code templates are available.

Code generation for an OR-node n. Given the adornments `{name, type, vars, bind, sel, filter}`, the following tasks are performed:

(a) Wrap the selections around the codes of the son nodes.

(b) Apply the dynamic filters.

(c) Perform the union.

Therefore, the code template has the following shape:

```
template_union(
    template_filterselect(
        template_select (code_son_1, sel_1), filter_1)
    . . .
    template_filterselect(
        template_select (code_son_k, sel_k), filter_k))
```

Code generation for an AND-node n. Given the adornments `{r-id, join, antijoin, semijoin, sel, proj, sip}`, the following tasks are performed:

(a) Perform sideways information passing (note that here the evaluation order of son nodes is important).
(b) Apply the join (`thetajoin, antijoin, semijoin`).
(c) Apply the selection.
(d) Perform the projection.

Code generation for an LFP-node n. The codes, generated for the k son nodes, are used as a k-dimensional fixpoint equation and are supplied to the `iterate`-operator.

Beyond stratification. As already mentioned, DECLARE may be configured such that, in case of a stratification violation, the query may still be evaluated. There are many interesting problems leading to non-stratified, but locally stratified programs. An example is the familiar parts-explosion program; a DECLARE solution can be found in Kießling and Güntzer (1990). DECLARE can accept such programs at the user's responsibility. Code generation follows exactly the scenario described before. Thus, we can offer a *default operational semantics* for non-stratified programs which, of course, has to be treated carefully.

5.6 Case Studies

Now we provide three case studies to show how the ORC works.

Uncle: (`parent, sex`, and `married` are base relations)

```
DEFINE VIRTUAL RELATION uncle(x string, u string) {
  IF   parent(x, pa), parent(pa, gpa), parent(u, gpa),
       sex(u, "male") WITH u /= pa
  THEN uncle(x, u);

  IF   parent(x, pa), parent(pa, gpa), parent(a, gpa),
       sex(a, "female"), married(a, u) WITH a /= pa
  THEN uncle(x, u); }

DEDUCE u FROM uncle("michael", u);
```

Assume that the ORC performs the following optimizations:

Figure 5. ERA-code for `Uncle`

```
-----------------------------------------------------------------
c_era("semijoin(parent, (1), (1),
                semijoin(parent, (0), (1),
                         select(parent, const_equal(0, 'michael'))),
                :select attr_unequal(0:0, 1:0))",
     &tmpRel[0]);
c_era("union(semijoin(tmpRel[0], (0), (0), sex,
                      :select const_equal(1:1, 'male'),
                      :project (0)),
             semijoin(semijoin(married, (0), (0), tmpRel[0]),
                      (0), (0), sex,
                      :select const_equal(1:1, 'female'),
                      :project (1)))",
     &QueryResult);
-----------------------------------------------------------------
```

- `Common-subexpression`: The three `parent`-atoms are joined only once.
- `Push-selection`: `"michael"` is pushed to the leaves of the QEP.
- `Join-order`: According to the bound-is-easier strategy, the `parent`-atom with `"michael"` as selection is the starting point for the three-way join.
- `Push-projection`, `specialize-operation`.

The resulting QEP is shown in Figure 11 of the Appendix. The generated ERA-code is shown in Figure 5.

The second `c_era` expression computes the result of the query using the common subexpression (`tmpRel[0]`) which is the result of the first `c_era` code. The `c_era` function parses its first argument and recursively calls the function `c_era` with the code of the sons (e.g., arguments 1 and 4 in the semijoin case). Then the results of these calls are combined and the final result is returned in the second argument of the `c_era` function.

Lucky_Man: (`human`, `sex`, `disease`, `parent`, and `married` are base relations)

```
DEFINE VIRTUAL RELATION lucky_man(x string) {
 IF human(x), sex(x, "male"),
    NOT (disease(x, ca) WITH ca IN ("cancer", "aids")),
    FORALL d (parent(d, x), sex(d, "female") IMPLIES married(d, _))
 THEN lucky_man(x); }

DEDUCE x FROM lucky_man(x);
```

Figure 6. ERA-code for `Lucky_man`

```
----------------------------------------------------------------------
c_era("antijoin(
          antijoin(semijoin(human, (0), (0), sex
                        :select const_equal(1, 'male')),
               (0), (0),
               select(disease, const_member(1,('cancer','aids')))),
          (0), (1),
          antijoin(semijoin(parent, (0), (0), sex
                        :select const_equal(1, 'female')),
               (0), (0), married))",
      &QueryResult);
----------------------------------------------------------------------
```

Since $\forall x(A(x) \rightarrow B(x)) \equiv \neg\exists x(A(x) \wedge \neg B(x))$, and because we enforce the covered variable restriction for `FORALL`-formulas, these two negations can safely be mapped to nested antijoins. Then let us assume that the ORC performs the following optimizations:

- `Join-order`: Start by joining positive atoms, then add the negative literal and the `FORALL`-formula via antijoins.
- `Push-projection, specialize-operation.`

The code produced from the resulting QEP, as shown in Figure 12 of the Appendix, is given in Figure 6.

Same_Generation: (`parent` and `human` are base relations)

```
   DEFINE VIRTUAL RELATION sg(x string, y string) {
      IF human(x) THEN sg(x, x);
      IF parent(x, x1), sg(x1, y1), parent(y, y1) THEN sg(x, y); }

   CONFIRM sg("julia", "werner");
```

The initial QEP for this folklore example is shown in Figure 13 of the Appendix. Now assume that the ORC performs the optimizations `supplementary-magic-set`, `join-order` and `specialize-operation`. From the resulting QEP (Figure 14 of the appendix), the ERA-code in Figure 7 can be generated. Here the first `iterate` computes the supplementary and minimagic sets, which are defined by mutually recursive rules. The second `iterate` computes the recursive predicate `sg` by using the supplementary and minimagic sets. It's worth pointing out here that the generated code, even for magic-set transformations, is amazingly compact.

Figure 7. ERA-code for `Same_Generation`

```
------------------------------------------------------------------
c_era("iterate(supmagic = equijoin(equijoin(parent,0,0,minimagic),
                                   3, 0, parent,
                                   :project (0:0, 1:0, 0:1, 1:1)),
            minimagic= union(project(supmagic, (2, 3)),
                                 create_temp_rel(
                                   'char a1[40]; char a2[40]',
                                   :init_value ('julia', 'werner'))))",
    &tmpRel[0], &tmpRel[1]);
c_era("iterate(sg = union(semijoin(tmpRel[0], (2,3), (0,1),
                                    sg, :project (0, 1)),
                          semijoin(select(tmpRel[1], attr_equal(0, 1)),
                                    (0), (0), human)))",
    &tmpRel[2]);
c_era("select(tmpRel[2],
           and(const_equal(0, 'julia'), const_equal(1, 'werner')))",
    &QueryResult);
------------------------------------------------------------------
```

6. The MVV-Expert

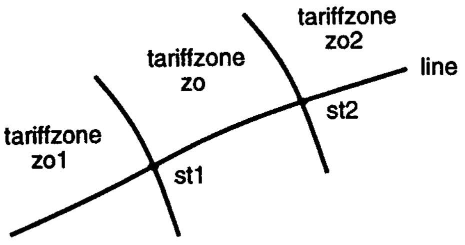
The *MVV-Expert* (Strauß, 1986; Bayer et al., 1987) is an expert system developed for the public transportation system of the Munich area to inform about connections, fares, and departure and arrival times.[2]

The Münchener Verkehrs- und Tarifverbund (MVV) is a network of all public transportation systems in the Munich area, which has about 2.3 million citizens. The MVV includes suburban trains, underground trains, streetcars, and buses serving a total of about 2,800 stations.

The MVV uses an integrated tariff system allowing a customer to ride on all its lines with only one ticket. Most of the tariff rules defining the fare of a ride from one station to another are displayed only graphically to the user. For this purpose, the MVV offers two maps that present an abstract view of the Munich area. These maps are the *Bartarifschemaplan* for computing the price of a single ride and the *Zeitkartentarifschemaplan* for the price of season tickets. Both maps separate the area into different tariff zones. The price of a ride depends on the number of zones crossed. The translation of this graphically based tariff system to a declarative description proved difficult and generated a complex rule system. Let

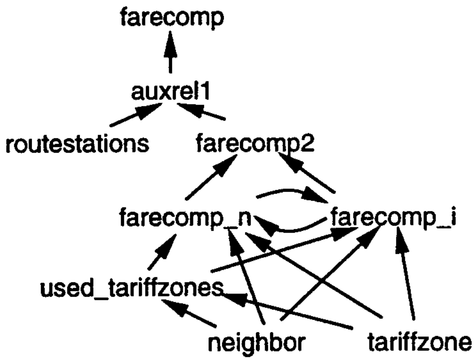
2. This system was designed and implemented as an early pilot application from late 1985 through 1987 and its first version was presented at the *Systec-fair* in 1986 in Munich and at the *Hannover-fair* in 1987.

Figure 8. Detail of Bartarifschemaplan map



```
IF   tariffzone(st1, zo),  tariffzone(st2, zo), neighbor(st1, st2),
       tariffzone(st1, zo1), tariffzone(st2, zo2)
       WITH zo2 /= zo, zo1 /= zo, zo1 /= zo2
   THEN used_tariffzone(st1,st2,zo);
```

Figure 9. Predicate connection graph for fare computation



us give an example for the conversion of one of the *Bartarif*-rules from the graphic to the logical formalism.

Figure 8 presents a detail of the Bartarifschemaplan and a rule deduced from the graphical representation. This rule illustrates that a ride including station `st1` and its neighbor station `st2` has to use zone `zo`, even though `st1` and `st2` also belong to other zones `zo1` and `zo2`. The predicate connection graph in Figure 9 gives an impression of the variety and complexity of virtual relations needed to count the number of tariff zones for an MVV-ride.

Altogether, the MVV-Expert consists of 13 base relations and 112 rules defining 40 virtual relations. Eight of these rules use negation, 51 rules are direct-recursive, 6 rules are mutually recursive, and 4 rules use aggregation. The smaller base relations (12 out of 13) were held in a local Relational-Lisp database. The largest base relation, containing about 10% of the full timetable information, consisted of approximately 40,000 tuples stored in an external Oracle database and connected via the database adaptor mechanism. This pilot application ran on two different hardware systems, on a Xerox 1108 Interlisp-D workstation and on an 80386/16Mhz machine. All DECLARE rules were optimized and compiled into Relational-Lisp. Typical queries to the MVV-Expert showed the following average response times:

- Search the shortest connections between two arbitrary stations: 1 sec.
- Show the fare for a tour: 1 - 4 sec.
- Show the cost of a season ticket: 4 - 10 sec.
- Show the arrival time for a certain connection: 3 - 5 sec.

Only queries for arrival and departure times access the external Oracle database. Each such access can be split into three components:

- Generate a SQL query and transmit it to Oracle: 0.1 sec.
- Execute the query on Oracle: 0.6 sec.
- Send the results back to Relational-Lisp, including necessary format transformations: 0.1 sec.

We learned a great deal from this early exposure to a realistic application. We were surprised by the good response times which came very close to a usable system. On this outcome nobody would have bet because of the overly complicated tariff system. We also learned about the difficulty of transforming graphical rules into "vanilla" logic programs without the support of adequate geometric data types.

Encouraging experiences have also been made by other in-house applications (e.g., a demo for inventory control, an air-travel study, and a leads-qualification system for marketing and sales support using a large external database) and from a pilot installation of DECLARE and SDS at a major company in Japan, where various industrial applications in the area of production control have been prototyped.

7. Productization and Commercialization

Initially, company strategy regarded CommonLisp not only as a technically suitable platform for DECLARE and SDS, but also as a marketable one. At that time a growing number of expert system applications (the marketable spin-offs of AI research) seemed to open the field for new technologies implemented in Lisp. As it turned out, however, a strong resistance against Lisp-based systems showed up among potential customers, because CommonLisp application programmers were not readily available. Consequently, application development had to take place in environments familiar to users of relational databases. This necessitated the migration of SDS into a conventional programming environment used for E-SQL and C-applications, and required that DECLARE be embedded into the C-language rather than CommonLisp.

7.1 Transition to the C-World

A slight transformation of the DECLARE syntax concerning the position of parentheses and other syntactic sugar was probably the easiest part of this transition. DECLARE was embedded into the C programming language in the same style as Embedded SQL, which SDS actually used as a substitute for the update features of Relational-Lisp. The definition of virtual relations using DECLARE rules and the formulation of queries is embedded into so-called *reasoning sections*. These sections can be defined in separate source files. A specific precompiler extracts the reasoning sections and passes them to the rule compiler ORC. The access to the results of a DECLARE query is achieved through the cursor mechanism of E-SQL. DECLARE cursor definitions are transformed into SQL cursor definitions for the application program. A complete re-implementation of Relational-Lisp in C, however, would have needed much more time than the original Lisp implementation. As a welcome alternative, Relational-C was implemented in cooperation with *TransAction Systems GmbH*, a Munich-based system house for relational database software. TransAction Systems upgraded the kernel of their proprietary SQL database system *TransBase* with the necessary extensions:

- A least fixpoint operator to compute recursive virtual relations.
- A representation of lists using strings.
- Intermediate temporary relations.
- A call mechanism for host language functions embedded in DECLARE rules.

The rule compiler ORC itself remained in CommonLisp, but this was completely hidden at the interface level. As DECLARE is a static compile-time language, the runtime system for a completed application did not need Lisp at all. Thus, we could essentially migrate DECLARE to the C-world by porting the runtime system from CommonLisp to C.

Some other issues dealing with existing relational databases merit more study.

Null values from SQL. Relations imported from existing external SQL-databases will inevitably contain null values, which now may participate in the deduction process. But current *Datalog*-systems are designed to deal with 2-valued logic only. The DECLARE system circumvented the well-known semantic traps simply by replacing each null value with a user-defined default value for the corresponding attribute type. The consequences of such a choice, however, cannot easily be determined in a complex deductions process, particularly when negation and grouping are involved.

Rule management. A crucial decision was whether to store rules inside the deductive database system. From an academic point of view, managing rules by the deductive database system itself yields the advantages of sharing not only data, but also the entire rule-based knowledge. In a marketing-driven situation, however, quite different arguments prevail. Prospective customers from the conventional data processing enterprise, accustomed to imperative programming, already suffer a sort of culture shock on their first confrontation with the declarative rule-based paradigm. This holds even for SQL programmers who regularly program in E-SQL. To mitigate this culture shock and to disturb their usual program development habits as little as possible, the conservative approach was to treat rules like ordinary programs, which are to be kept in files. This does not interfere with software engineering tools like make-files, etc., which are in action everywhere. Consequently, an early prototype of a rule management component as part of DECLARE was deliberately abandoned in favor of the traditional programming approach.

7.2 Software-Engineering Aspects

DECLARE and SDS were engineered according to commercial practice for high-quality software development. Far more than 50 person years were invested into research and development time, let alone the expenditure for marketing and sales. At peak times the head count of the DECLARE team in Munich reached 16 R&D people. One of the most challenging aspects of this project was the need to provide a complete product to stand out in the marketplace, while the theory and necessary methods were evolving. To keep up with this process the development operation had to be laid out with special care. A relatively large portion of development time was spent on design and specification phases. The use of abstract data types and the separation of modules communicating only through specified interfaces proved particularly beneficial. This strict modularity and thorough documentation, not only of final results but also of the reasons behind design decisions, allowed the complete re-engineering of implementation modules, if required. Three reasons sometimes appearing together made such an effort necessary from time to time: addition of new features and optimization techniques, practical experiences with the current implementation, and shifting marketing requirements.

A variety of tools was provided to inspect the global internal data structures of the ORC (e.g., tools to visualize and browse the PCG and the QEP). Concerning quality assurance, a test suite was implemented for automated regression testing of the entire DECLARE system and of single modules. Considerable manpower was put into test tools and test specifications. A steadily growing collection of test cases was generated to ensure the correctness of the entire system after each software change or extension. In addition to mere verification, the test tool was extended to collect sample performance data on the effect of program changes. For instance, the ERA-code was systematically tuned, paying special attention to low-level details like cons-consumption in CommonLisp. Also, after having observed from our applications that deductions often involve a large number of ERA-operations on rather small (intermediate) relations, hybrid algorithms for many ERA-operations were implemented, aiming to cut down the overhead for very small relations.

This systematic approach enabled us to compare different optimizer configurations in detail. Implementation of the system itself was performed in several stages, characterized by an early realization of a basic operational system to which optimization techniques and strategies were added incrementally. This allowed the early gathering of practical experiences. Some complex examples, like the MVV-expert, accompanied most of the development process.

8. Adding Heuristic Knowledge and Control

Now we report some results from applying DECLARE to problem areas that have not yet received much attention. Current deductive database systems offer a purely declarative framework; that is, the user specifies *what* results to compute, but the user can hardly affect *how* these results are computed. Narrowing the search space by clever methods like magic-sets is not always sufficient. But the user sometimes has additional heuristic knowledge for computing the answer to a query.

In Section 8.1 we present theoretical and experimental work on combinatorial problems from the DECLARE project, not released with the product and done by only a few people from the DECLARE team. Section 8.2 addresses a general framework for an expert database system that was not prototyped.

8.1 Combinatorial Problems in Deductive Databases

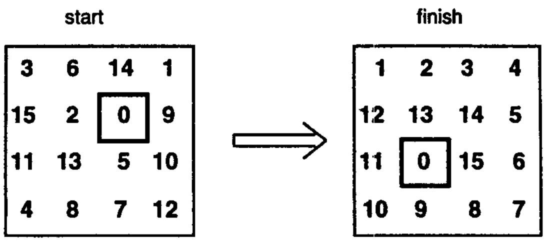
Extending *Datalog* to *Datalog*$^{neg+func}$ brings us into the realm of NP-complete problems. Solving these by breadth-first search algorithms (i.e., semi-naive iteration) or by depth-first search (i.e., SLD-resolution) is not feasible. Instead, best-first search algorithms like A* must be used. The A*-algorithm was originally introduced in AI to efficiently solve difficult search and planning problems that produced a combinatorial explosion. As shown by Schmidt et al. (1989), a generalized version of A*, called the *DBA*-algorithm*, can be implemented using the so-called *sloppy-delta iteration* scheme for differential goal-directed deduction introduced by Güntzer et al. (1987) and Schmidt et al. (1987). Let us demonstrate its effectiveness by reporting some benchmark results gained from the familiar 15-puzzle problem.[3] The *15-puzzle* consists of 15 numbered, movable tiles set in a 4x4-frame. One cell of the frame is always empty, making it possible to move an adjacent numbered tile into the empty cell or to move the empty cell. The goal is to find a series of moves that end up in a predefined ordering of the tiles in the 4x4-frame.

The deductive database consists of the virtual relation `puzzle(state,row,col)`. `State` is a puzzle represented by a list of lists (with indices starting from 0). `Row` and `col` give the position of the empty cell in `state`. Given a puzzle state, `up`, `down`, `left` and `right` are user-defined functions that move the empty cell into the specified direction. The DECLARE rules, a sample start position, and a predefined goal position are depicted in Figure 10.

To apply A*, heuristics estimating the cost of a path from a puzzle state to the

3. The 15-puzzle is used here for demonstration purposes only, and is not intended to be a genuine deductive database application.

Figure 10. 15-puzzle in DECLARE



```
DEFINE VIRTUAL RELATION
   puzzle (p list, row integer, col integer) {
      ASSERT puzzle((( 3,  6, 14,  1), (15, 2, 0,  9),
                     (11, 13,  5, 10), ( 4, 8, 7, 12) ), 1, 2);
      IF     puzzle(p, row, col) WITH row /= 0
      THEN   puzzle(up(p), row-1, col);
      IF     puzzle(p, row, col) WITH row /= 3
      THEN   puzzle(down(p), row+1, col);
      IF     puzzle(p, row, col) WITH col /= 0
      THEN   puzzle(left(p), row, col-1);
      IF     puzzle(p, row, col) WITH col /= 3
      THEN   puzzle(right(p), row, col+1); }

CONFIRM puzzle((( 1, 2,  3, 4), (12, 13, 14, 5),
                (11, 0, 15, 6), (10,  9,  8, 7) ), 2, 1);
```

goal state is supplied.[4] Using an experimental version of DECLARE containing the DBA*-algorithm, we succeeded in solving randomly generated puzzles at an average time of about 80 seconds on a slow 80386/16Mhz machine. As in Kießling (1992) for the N-queens benchmark, we invite everybody to beat this result. Further extensions, implementing the BS*-algorithm for heuristic bidirectional search by fixpoint iteration with *subsumption*, can be found in Köstler et al. (1993).

We experimented with other extensions in the area of *monotonicity* information. For example, consider a quadratic recursion with a fixpoint equation:

$F(S) = S*S + R$

Normally, the differential expression required for the semi-naive iteration is:

$dF(S) = \Delta*S + \Delta*\Delta + S*\Delta$

Under certain monotonicity assumptions it can be simplified considerably to:

$dF(S) = \Delta*\Delta$

4. We used the Manhattan distance heuristics and the sequencing score heuristics (Schmidt et al., 1989).

The solution to the N-queens problems in Kießling (1992) describes such a case.

8.2 Expert Database System Architecture

Schmidt (1992) proposed an *expert database system* architecture that offers a general framework to the user for specifying application-specific control knowledge. This framework distinguishes between an *object-level* and a *meta-level*. At the object-level, the user specifies the logical aspects of a problem by a conventional deductive database language like DECLARE. At the meta-level, the user provides another deductive database containing control information about the object-level deduction process. The feedback from the meta-level to the object-level is specified by control predicates, which allow us to discard irrelevant tuples or rules, to prefer useful tuples or rules, and to prematurely terminate recursion, if desired.

Let us discuss some features of this architecture by the Air-Travel Expert introduced by Freitag and Biernath (1988), which computes a flight connection between two airports:

Air-Travel Expert Database (Object-Level):

- `SR(f, g, h)`: `h` is the smallest region geographically containing `f` and g.
- `SubR(g, h)`: The region `h` geographically contains the region g.
- `DF(x, y)`: There is a direct flight between x and y.
- `FC(x, y, r, g)`: There is a flight connection between x and y on the route `r`, and g is the smallest geographic region containing all stops of this flight. This predicate is defined by the following rules.
- A direct flight from x to y implies a flight connection from x to y:
  ```
  IF DF(x, y), SR(x, y, h) THEN FC(x, y, [y, x], h);
  ```
- If there is a flight connection from x to z and a direct flight from z to y, then there is a flight connection from x to y:
  ```
  IF   FC(x, z, r, f), DF(z, y), SR(f, y, h), NOT Member(y, r)
  THEN FC(x, y, [y | r], h);
  ```

Without any control, the bottom-up evaluation of this program generates a much too large search space. Common heuristics for this application, however, is not to leave the smallest geographic region, which contains both the start airport and the destination airport.[5] Again the sloppy-delta iteration scheme mentioned above can be used to prune the search space. The conditions that select the relevant tuples are specified at the meta-level by the control predicate `Prefer`(q, (([q_1], n_1),. . ., ([q_k], n_k))), where q_i ($1 \leq i \leq k$) are object-level conditions and the optional value $n_i \in \mathbb{N}$ specifies the number of iteration steps:

5. E.g., when looking for a flight connection from Augsburg, Germany to London, consider only those flights not leaving Europe.

In the first n_1 iteration steps prefer those tuples (deduced up to now) that fulfill $q_1, \ldots$, in the next n_k iteration steps prefer those tuples (deduced up to now) that fulfill q_k. (If n_i is not given, the condition q_i is used until no more tuples can be deduced.)

Using this control predicate, we can specify the above heuristics by the following meta-rule.[6]

Air-Travel Expert Database (Meta-Level):

If a query @q is looking for flight connections from @a to @b, and if @h is the smallest region containing both @a and @b, then don't leave @h.

```
Prefer( @q, ([FC(_,_,_,g), SubR(g, @h)])) ←
    Query ( @q, [FC( @a, @b, _, _)]), SR( @a, @b, @h).
```

Altogether, we claim that deductive databases need a handle for user knowledge. This view is also expressed by McCarthy (1987): "Reasoning and problem-solving programs must eventually allow the full use of quantifiers and sets, and have strong enough control methods to use them without combinatorial explosion." Such handles can be employed to enter heuristics, subsumption, or monotonicity information. Currently CORAL (Ramakrishnan et al., 1992) with its annotation mechanism already offers such features to a certain extent.

9. Summary

For every new idea there is always the dilemma of gain and pain to be first in the marketplace. While prospective customers were excited by the technical potential of DECLARE and SDS, their most recent (not yet depreciated) investments in relational technology prevented many from making a change. The lack of textbooks and *Datalog* programmers five years ago was certainly another retarding factor. Concerning Lisp vs. C, we found that—despite the bad reputation of Lisp-based systems—the efficiency of DECLARE/Lisp proved quite high (only the infamous garbage collector was felt disturbing at times). Thus, the choice of C as a host environment for DECLARE and SDS is more a matter of marketing. From our own experience and customer feedback it became apparent that the current *Datalog* model needs to be extended, most urgently towards non-stratified recursion (including aggregates as it happened lately), abstract data types, modules and handles to enter user knowledge for semantic optimizations and heuristic control. Moreover, dealing with incomplete knowledge (nulls) is a practical issue.

6. For the meta-level we use a general Horn-clause syntax instead of DECLARE syntax. For distinguishing meta-level variables from object-level variables, meta-level variables start with @.

In 1989, DECLARE and SDS were certainly unique when compared with other available prototype systems. Concerning our modular and extensible optimizer architecture, we have nothing but positive experiences to report. During the development the contemporary work done for LDL has caught most of our attention.

Although never reaching a commercial breakthrough (a not so rare fate for start-up technologies) we believe that DECLARE and SDS have provided a feasibility proof for deductive database technology in full scale, even in a heterogeneous multi-database environment with global schema integration features. But deductive database technology as it stands does not seem to provide a full quantum leap justifying a major switch-over in industry. Nevertheless, it will make its way, if not in its present "Spartan" form, then enriched with the necessary ingredients for cost-effective application development and maintenance.

Acknowledgments

Many people have contributed to DECLARE and SDS. In particular we would like to thank the former members of the DECLARE team in Munich: Rüdiger Azone, Olaf Biernath, Maria Friedrich, Lothar Kappen, Franz Kronseder, Eva Krüger, Benedikt Nuiding, Elisabeth Peischl, Rudi Raith, Josef Schreiner, Johannes Schwaiger, Petra Schweier, Nikolaus Steger, Helmut Thöne, Andreas Wittkowski, Arne Wolfsteiner, Oliver Wörl and Christoph Zischgl. Likewise we thank Rudolf Bayer, who acted as strategic consultant throughout the DECLARE and SDS project.

References

Apt, K., Blair, H., and Walker, A. Towards a theory of declarative knowledge. In: Minker, J., ed. *Foundations of Deductive Databases and Logic Programming.* San Mateo, CA: Morgan Kaufmann, 1987, pp. 89–148.

Balbin, J. and Ramamohanarao, K.L. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming,* 4(3):259–262, 1987.

Bancilhon, F. Naive evaluation of recursively defined functions. In: Brodie, M., and Mylopoulos, J. eds. *On Knowledge Base Management Systems.* Berlin: Springer, 1986, pp. 165–178.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. Magic sets and other strange ways to implement logic programs. *Proceedings of the Fifth ACM Symposium on the Principles of Database Systems,* Cambridge, MA, 1986.

Bayer, R. Database Technology for Expert Systems. *GI Congress Wissensbasierte Systeme,* Informatik Fachberichte 112, Springer, 1985, pp. 1–16.

Bayer, R., Güntzer, U., Kießling, W., Strauß , W., Obermaier, J. Deduktions- und Datenbankunterstützung für Expertensysteme. *GI Congress Datenbanksysteme in Büro, Technik und Wissenschaft,* Darmstadt, April 1987, pp. 1 - 16.

Beeri, C., Ramakrishnan, R. On the power of magic. *Proceedings of the Sixth ACM Symposium on the Principles of Database Systems,* San Diego, CA, 1987.

Ceri, S., Gottlob, G., and Tanca, L. *Logic Programming and Databases*. Berlin: Springer, 1989.

Freitag, B. and Biernath, O. An Airtravel Expert Database. *Proceedings of the Third International Conference on Data and Knowledge Bases*, Jerusalem, 1988.

Freitag, B., Schütz, H., and Specht, G. LOLA—A logic language for deductive databases and its implementation. *Proceedings of the Second International Symposium for Advanced Applications*, Kyoto, Japan, 1991.

Gallaire, H., Minker, J., and Nicolas, J.-M. An overview and introduction to logic and databases. In: Gallaire, H. and Minker, J., eds. *Logic and Databases*, New York: Plenum Press, 1978, pp. 123—134.

Gallaire, H., Minker, J., and Nicolas, J.-M. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, 1984.

Güntzer, U., Kießling, W., and Bayer, R. On the evaluation of recursion in (deductive) database systems by efficient differential fixpoint iteration. *Proceedings of the Third International Conference on Data Engineering*, Los Angeles, 1987.

Hillman, A. *RelationalLisp User's Guide*. San Jose, CA: MAO Intelligent Systems, 1988.

Kießling, W. Dynamic filtering. In: Ozkarahan, E., ed. *Database Machines and Database Management*, Englewood Cliffs, NJ: Prentice Hall, 1986, pp. 312–317.

Kießling, W., Wittkowski, A., Schmidt, H., Strauß, W., and Azone, R. DECLARE: A deductive database language for large knowledge-based applications. MAD Intelligent Systems GmbH, Munich, unpublished internal paper, 1987.

Kießling, W. and Güntzer, U. Deduktive Datenbanksysteme auf dem Weg zur Praxis. *Informatik Forschung und Entwicklung (Special Issue on Non-Standard Database Systems)*, 5:177–187, 1990.

Kießling, W. A complex benchmark for logic programming and deductive databases, or who can beat the N-Queens? *SIGMOD Record*, 21(4):28–34, 1992.

Köstler, G., Kießling, W., Thöne, H., and Güntzer, U. The differential fixpoint operator with subsumption. *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, AZ, 1993.

McCarthy, J. Generality in artificial intelligence. *Communications of the ACM*, 30(12):1030–1035, 1987.

Morris, K., Ullman, J.D., and van Gelder, A. Design overview of the NAIL! system. *Proceedings of the Third International Conference on Logic Programming*, London, 1986.

Mumick, I., Finkelstein, S., Pirahesh, H., and Ramakrishnan, R. Magic is relevant. *Proceedings of the ACM SIGMOD Conference on Management of Data*, Atlantic City, NJ, 1990.

Naqvi, S. and Tsur, S. *A Logical Language for Data and Knowledge Bases*. Rockville, MD: Computer Science Press, 1989.

Phipps, G., Derr, M., and Ross, K. Glue-NAIL!: A deductive database system. *Proceedings of the ACM SIGMOD Conference on Management of Data*, Denver, CO, 1991.

Przymusinski, T. On the declarative semantics of deductive databases and logic programs. In: Minker, J., ed. *Foundations of Deductive Databases and Logic Programming*, San Mateo, CA: Morgan Kaufmann, 1987, pp. 193–216.

Ramakrishnan, R., Beeri, C., and Krishnamurthy, R. Optimizing existential datalog queries. *Proceedings of the Seventh ACM Symposium on the Principles of Database Systems*, Austin, TX, 1988.

Ramakrishnan, R., Srivastava, D., and Sudarshan, S. CORAL: Control, relations and logic. *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, Vancouver, 1992.

Sacca, D. and Zaniolo, C. Implementation of recursive queries for a data language based on pure horn logic. *Fourth International Conference on Logic Programming*, Melbourne, Australia, 1987.

Schmidt, H. Implementierung von Delta-Iterationsverfahren in R-Lisp. Diploma thesis, Technical University of Munich, November, 1986.

Schmidt, H., Kießling, W., Güntzer, U., and Bayer, R. Compiling exploratory and goal-directed deduction into sloppy delta-iteration. *Proceedings of the Fourth IEEE Symposium on Logic Programming*, San Francisco, 1987.

Schmidt, H., Kießling, W., Güntzer, U., and Bayer, R. DBA*: Solving combinatorial problems with deductive databases. *Proceedings GI/SI Congress Datenbanksysteme in Büro, Technik und Wissenschaft*, Zürich, March 1989.

Schmidt, H. Communicating control knowledge to a deductive database system. *Proceedings ACM Computer Science Conference*, Kansas City, March 1992.

Strauß, W. *Erweiterung des MVV-Experten in R-Lisp*. Diploma thesis, Technical University of Munich, November, 1986.

Tsur, S. and Zaniolo, C. LDL: A logic-based data-language. *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, 1986.

Ullman, J. Implementations of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1985.

Ullman, J. *Principles of Database and Knowledge-Base Systems*, Volume II, Rockville, MD: Computer Science Press, 1989.

Vaghani, J., Ramamohanarao, K., Kemp, D., Somogyi, Z., and Stuckey, P. Design overview of the Aditi deductive database system. *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, 1991.

van Emden, M. and Kowalski, R. Semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

van Gelder, A. A message-passing framework for logical query evaluation. *Proceedings of the ACM SIGMOD Conference on Management of Data*, Washington, DC, 1986.

Vieille, L., Bayer, P., Küchenhoff, V., and Lefebvre, A. EKS-V1, a short overview. *AAAI-90 Workshop on Knowledge Base Management Systems*, Washington, DC, 1990.

Zaniolo, C. The representation and deductive retrieval of complex objects. *Proceedings 11th International Conference on Very Large Data Bases*, Stockholm, 1985.

Appendix

The subsequent figures contain QEPs for the case studies performed in Section 5.6.

Figure 11. Optimized QEP for `Uncle`

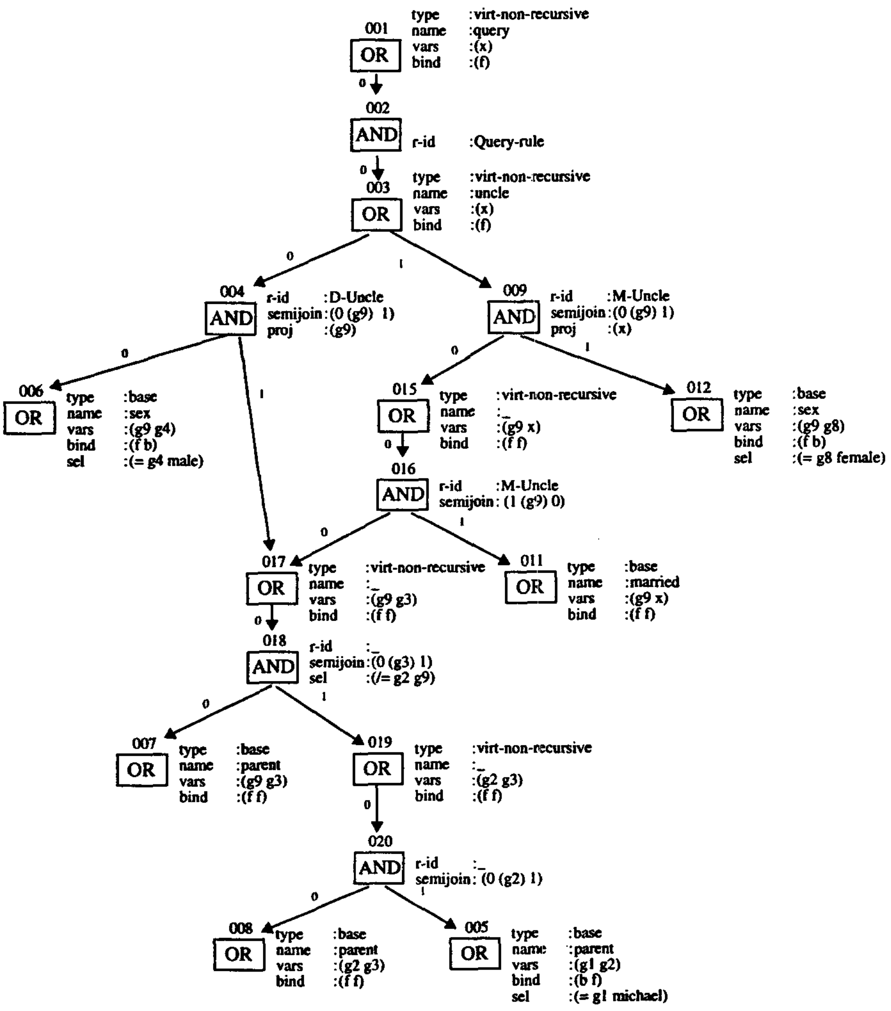

001 OR type :virt-non-recursive name :query vars :(x) bind :(f)
002 AND r-id :Query-rule
003 OR type :virt-non-recursive name :uncle vars :(x) bind :(f)
004 AND r-id :D-Uncle semijoin :(0 (g9) 1) proj :(g9)
009 AND r-id :M-Uncle semijoin :(0 (g9) 1) proj :(x)
006 OR type :base name :sex vars :(g9 g4) bind :(f b) sel :(= g4 male)
015 OR type :virt-non-recursive name :_ vars :(g9 x) bind :(f f)
012 OR type :base name :sex vars :(g9 g8) bind :(f b) sel :(= g8 female)
016 AND r-id :M-Uncle semijoin: (1 (g9) 0)
017 OR type :virt-non-recursive name :_ vars :(g9 g3) bind :(f f)
011 OR type :base name :married vars :(g9 x) bind :(f f)
018 AND r-id :_ semijoin :(0 (g3) 1) sel :(/= g2 g9)
007 OR type :base name :parent vars :(g9 g3) bind :(f f)
019 OR type :virt-non-recursive name :_ vars :(g2 g3) bind :(f f)
020 AND r-id :_ semijoin: (0 (g2) 1)
008 OR type :base name :parent vars :(g2 g3) bind :(f f)
005 OR type :base name :parent vars :(g1 g2) bind :(b f) sel :(= g1 michael)
0
1

Figure 12. Optimized QEP for `Lucky_man`

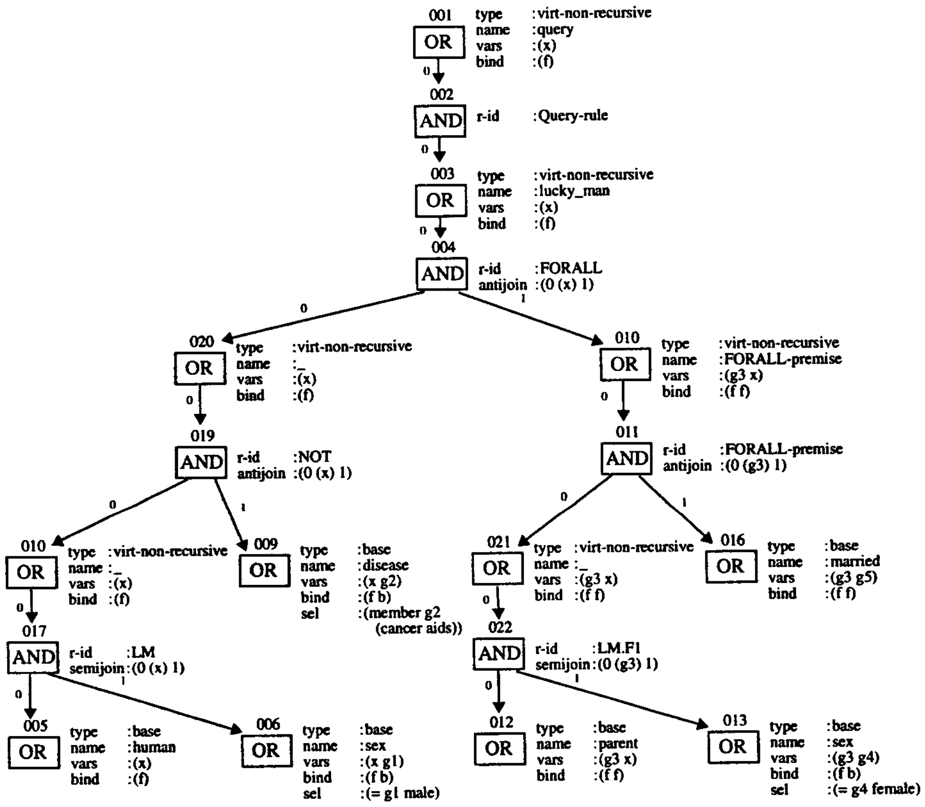
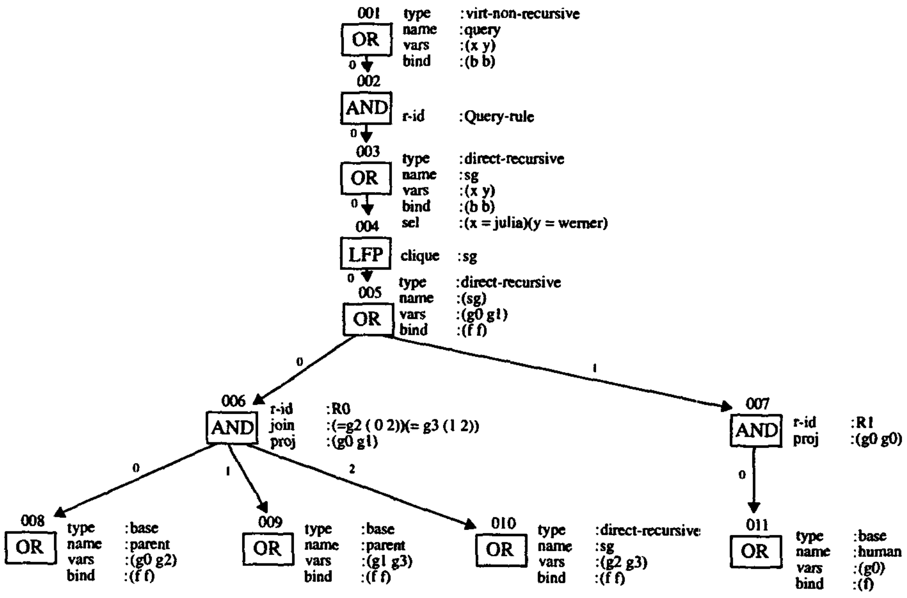
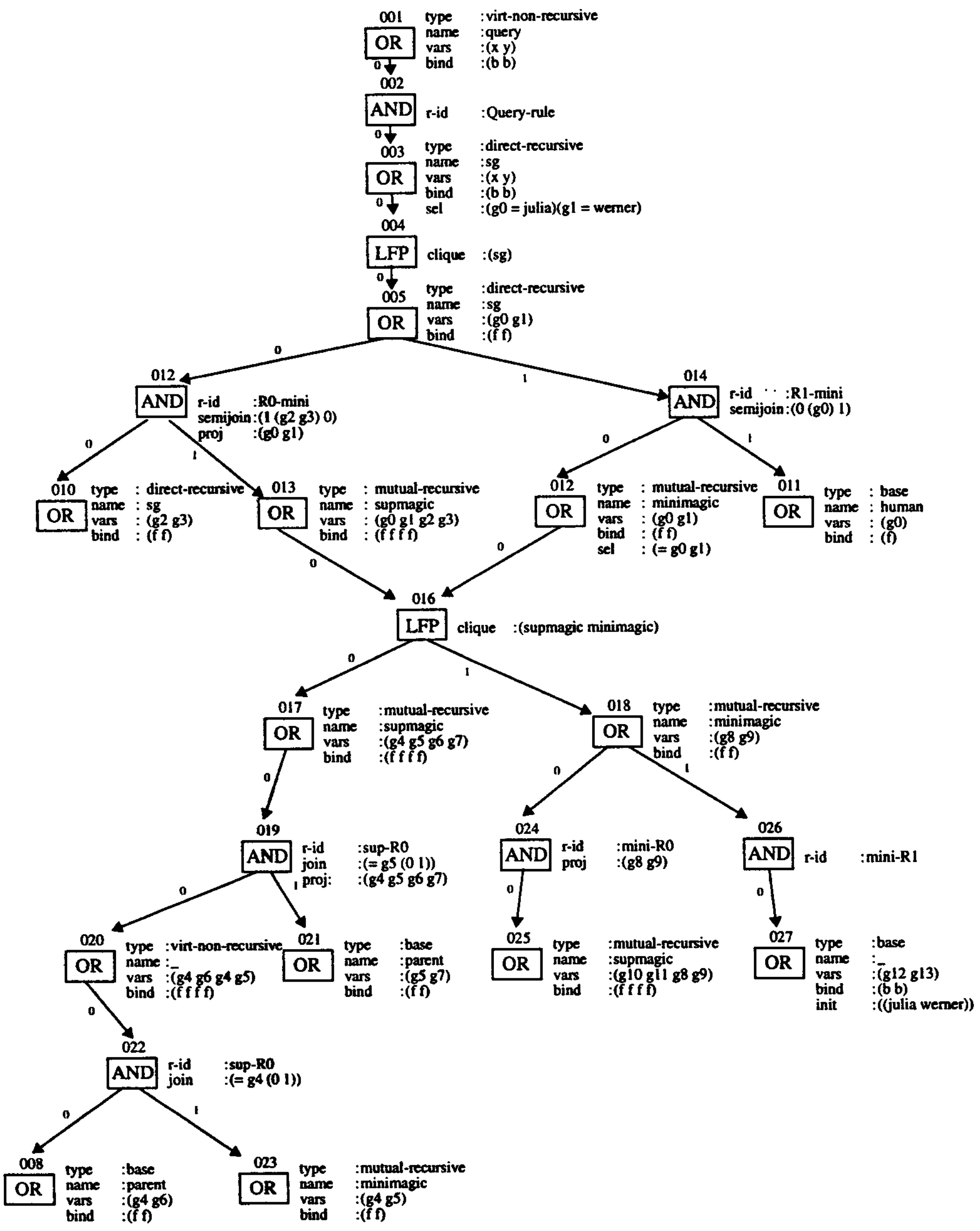

001 OR type :virt-non-recursive name :query vars :(x) bind :(f)
002 AND r-id :Query-rule
003 OR type :virt-non-recursive name :lucky_man vars :(x) bind :(f)
004 AND r-id :FORALL antijoin :(0 (x) 1)
020 OR type :virt-non-recursive name :_ vars :(x) bind :(f)
019 AND r-id :NOT antijoin :(0 (x) 1)
010 OR type :virt-non-recursive name :_ vars :(x) bind :(f)
017 AND r-id :LM semijoin :(0 (x) 1)
005 OR type :base name :human vars :(x) bind :(f)
006 OR type :base name :sex vars :(x g1) bind :(f b) sel :(= g1 male)
009 OR type :base name :disease vars :(x g2) bind :(f b) sel :(member g2 (cancer aids))
010 OR type :virt-non-recursive name :FORALL-premise vars :(g3 x) bind :(f f)
011 AND r-id :FORALL-premise antijoin :(0 (g3) 1)
021 OR type :virt-non-recursive name :_ vars :(g3 x) bind :(f f)
022 AND r-id :LM.F1 semijoin :(0 (g3) 1)
012 OR type :base name :parent vars :(g3 x) bind :(f f)
013 OR type :base name :sex vars :(g3 g4) bind :(f b) sel :(= g4 female)
016 OR type :base name :married vars :(g3 g5) bind :(f f)
0
1

Figure 13. Initial QEP for `Same_generation`



001 OR type :virt-non-recursive name :query vars :(x y) bind :(b b)
0
002 AND r-id :Query-rule
0
003 OR type :direct-recursive name :sg vars :(x y) bind :(b b) sel :(x = julia)(y = werner)
0
004 LFP clique :sg
0
005 OR type :direct-recursive name :(sg) vars :(g0 g1) bind :(f f)
0
1
006 AND r-id :R0 join :(=g2 (0 2))(= g3 (1 2)) proj :(g0 g1)
007 AND r-id :R1 proj :(g0 g0)
0
1
2
0
008 OR type :base name :parent vars :(g0 g2) bind :(f f)
009 OR type :base name :parent vars :(g1 g3) bind :(f f)
010 OR type :direct-recursive name :sg vars :(g2 g3) bind :(f f)
011 OR type :base name :human vars :(g0) bind :(f)

Figure 14. Optimized QEP for `Same_generation`


001 OR type :virt-non-recursive name :query vars :(x y) bind :(b b)
002 AND r-id :Query-rule
003 OR type :direct-recursive name :sg vars :(x y) bind :(b b) sel :(g0 = julia)(g1 = werner)
004 LFP clique :(sg)
005 OR type :direct-recursive name :sg vars :(g0 g1) bind :(f f)
012 AND r-id :R0-mini semijoin :(1 (g2 g3) 0) proj :(g0 g1)
014 AND r-id :R1-mini semijoin :(0 (g0) 1)
010 OR type : direct-recursive name : sg vars : (g2 g3) bind : (f f)
013 OR type : mutual-recursive name : supmagic vars : (g0 g1 g2 g3) bind : (f f f f)
012 OR type : mutual-recursive name : minimagic vars : (g0 g1) bind : (f f) sel : (= g0 g1)
011 OR type : base name : human vars : (g0) bind : (f)
016 LFP clique :(supmagic minimagic)
017 OR type :mutual-recursive name :supmagic vars :(g4 g5 g6 g7) bind :(f f f f)
018 OR type :mutual-recursive name :minimagic vars :(g8 g9) bind :(f f)
019 AND r-id :sup-R0 join :(= g5 (0 1)) proj: :(g4 g5 g6 g7)
024 AND r-id :mini-R0 proj :(g8 g9)
026 AND r-id :mini-R1
020 OR type :virt-non-recursive name :_ vars :(g4 g6 g4 g5) bind :(f f f f)
021 OR type :base name :parent vars :(g5 g7) bind :(f f)
025 OR type :mutual-recursive name :supmagic vars :(g10 g11 g8 g9) bind :(f f f f)
027 OR type :base name :_ vars :(g12 g13) bind :(b b) init :((julia werner))
022 AND r-id :sup-R0 join :(= g4 (0 1))
008 OR type :base name :parent vars :(g4 g6) bind :(f f)
023 OR type :mutual-recursive name :minimagic vars :(g4 g5) bind :(f f)
0
1