Architectural Considerations for Extending a Relational DBMS with Deductive Capabilities[1]

Michael Dobrovnik, Roland T. Mittermeir

Institut für Informatik
Universität Klagenfurt
Universitätsstraße 65-67
A-9020 Klagenfurt, Austria
e-mail: {michi,mittermeir}@ifi.uni-klu.ac.at

Abstract

This paper describes the development rationale and the architecture of a prototypical expert-database system. Knowledge processing capabilities of SQL were enhanced by extending the language by recursive views. This work is based on an evolutionary approach; smooth integration with the base language was an important development aim.

After a discussion of the main design alternatives, the architecture of a prototype is presented. Finally the progress of the project is described and possibilities for further extension are indicated.

1 Recursive Views

1.1 Motivation

A host of modern applications demand knowledge processing capabilities in combination with the support of large scale volume data processing capabilities and multi-user support for concurrent access and flexible combination of persistent information as provided by todays data base systems. But classical expert system shells lack important features needed in conjunction with bulk transaction processing, support for persistence, and integrity preservation over long spans of time. Hence, systems supporting multi-paradigm applications become increasingly important.

[1]The work on this project was partly supported by the Austrian Fonds zur Förderung der wissenschaftlichen Forschung under Contract P6772P.

At the time this project started, various options to achieve the above aim have already been proposed in the literature (see e.g. [Gall81, Gall84, Brod86, Kers86, Wied 86]). They can be classified into three broad categories:

- extensions of logical programming languages or expert system shells by appropriate permanent storage management (back-end storage management);
- development of database systems with "logical" query languages;
- extensions of database systems by "reasoning facilities".

In the project on which we are reporting here, the latter approach had been adopted. However, we wanted to follow this approach in such a way that we could fully build SQL's high acceptance in the marketplace. To achieve this aim, a solid formal definition of certain SQL features became necessary before searching for an adequate linguistic and architectural design of such an extension. While the formal aspects have been reported already, this paper presents the architectural considerations which guided this project.

The choice for this approach has been founded on the consideration that relational database systems enjoy high penetration into a host of application areas. One reason for this success surely is the widespread use of standard database languages such as SQL [SQL86,Date87]. SQL can be characterized as an end user oriented, mainly declarative language which plays a central role in the database field, even in spite of its well known deficiencies [Date87].

One of the most important restrictions of SQL is its lack of computational completeness [Aho79]. So, an important class of systems such as knowledge based systems or decision support systems, but also technical systems demanding special search characteristics [Boud92], are not well supported. A particular reason for this deficiency is that recursive problems cannot be adequately attacked by means of standard SQL. But recursion plays an important role in deductive systems. Two of the most prominent textbook examples for this class of problems are path problems and bill of material calculations. Hence, the main idea of the XPL*SQL-project was to extend the capabilities of SQL in such a way that the extended language provides good support for a broad range of recursive problems.

The linguistic mechanism we needed for obtaining our aim was the well known view mechanism. It allows to create virtual relations by declaring a rule that describes how to compute them. The view construction mechanism has been extended to support **recursive views**.

1.2 General Transitive Closure

The transitive closure T of a relation R is defined as [Eder90a]:

$$LFP(T = union(R,COMP(R,T)))$$

COMP means composition and is an equijoin where the join-attributes are eliminated by projection. The least-fixpoint operator *LFP* evaluates T to the smallest set, for which the equation is valid.

To demonstrate this concept, let us consider a binary relation *flight(from,to)*, which associates cities that can be reached with one single flight. This relation clearly is transitive, so it makes sense to compute the transitive closure *connection* of *flight*, which contains all flight connections between two cities, formally:

$$LFP(connection = union\ (connection, flight \bowtie_{flight.to=connection.from} connection))$$

It has to be pointed out, though, that the concept of transitive closure of a relation may not contain any attributes pertaining to the specific association just established. E.g. in the example, it is not possible to total the distance or the duration of connections. Certainly, this is a main disadvantage of pure transitive closure and makes it unsuitable for a large class of applications. Therefore, the concept was generalized [Eder90a,Eder90b] in the following way:

$$LFP(GT = union(R,COMPEX(R,GT)))$$

There, *R* is a base relation as before, *GT* is the generalized transitive closure. The main difference between transitive closure and general transitive closure lies in *COMPEX*. *COMPEX* stands for composition-expression and is a selection on the carthesian product of *R* and *GT*, combined with a projection which may also contain arithmetic expressions. The introduction of this composition-expression allows the definition of attribute values as computable functions, whereas the generalization from the equijoin to a selection on the carthesian product allows to formulate non-trivial conditions for linking tuples. An example for general transitive closure will be given in a subsequent section.

1.3 Integration of Generalized Transitive Closure into SQL

General transitive closure is a special form of a linear recursive deduction rule. When one considers SQL, there is a mechanism which allows for the definition of derived relations, which are better known as **views**. A view is a virtual relation whose extension is computed according to a declarative specification, the view definition, which can be seen as a deduction rule. Whereas one could argue that from such a perspective, SQL is a language with deductive components, there is one main shortcoming of views in standard SQL. The language explicitly forbids to reference the view to be defined in the definition part itself, i.e. recursion is not permitted.

Considering the fact, that views can be interpreted as nonrecursive deduction rules, and that views are a well understood feature of SQL which is broadly used in practice, it seems to be promising to extend the view concept and to explicitely allow the definition of **recursive views**. This evolutionary approach not only integrates very well with the basic language, it has as main advantage, that it does not require any change in the application pattern. Neither a user querying a view, nor any special tool (application generator, report writer, ...) using those views, need to take special consideration as to whether a view is defined recursively or in the usual way.

However, there are some minor deficiencies one has to bear in mind using recursive views. In general, recursive views may not be updated, queries on them can take longer to complete than on conventional views, and the results of a query may be infinite. While the first and second points are inherently connected with recursive views, the possibility for infinite results requires special treatment (see [Eder90a]).

Nevertheless, besides increasing the expressive power of the language, this specific approach meets some important criteria for extending a language [Mitt88]. The principle of recursive views is easy and safe to use and it incorporates a minimal number of new constructs. The new feature is orthogonal to existing language elements, it can be formally described, and it can be optimized to some extent.

1.4 Syntax of Recursive Views

The syntactical extensions of the definition of SQL are mainly captured in one single place, namely the *recursive-view-definition-statement* which is presented (in a slightly simplified form) in Figure 1. Other aspects of the language, notably the select statement, remained unchanged.

A simple example of the application of the new construct can be found in the appendix. Now we briefly give an informal description of some of the nonterminals mentioned in Figure 1. For a more thorough treatment, we refer to [Eder90a, Eder90b, and Dobr91].

The *attributed-column-list* extends the standard *column-list* of SQL. With *INC* and *DEC* respectively, the specification of monotonous characteristics of certain attributes is allowed. This information is crucial in optimization and assuring the finiteness of certain queries. The *set-type* specifies, whether a certain view should be treated as a set-relation, having only distinct tuples and where duplicates have to be eliminated, or as a multiset-relation, where duplicate tuples must be taken into account.

It should be noted that recursive views can be used as targets of queries like any other table or conventional view (with some minimal restrictions, see [Eder90a]). As small as the syntactical extensions to standard SQL for the definition of recursive views may be, the possibility to use recursive views in virtually all contexts where ordinary views are permitted implies that fundamental changes in the SQL-interpreter must be made.

statement ::= ... |
    create-view-statement |
    create-recursive-view-statement |
    ...

create-view-statement ::=
    CREATE VIEW viewname [(column-list)]
    AS SELECT [set-type] select-list
    FROM table-reference-list
    [where-clause]
    [group-by-clause] [having-clause];

create-recursive-view-statement ::=
    CREATE VIEW viewname ***(attributed-column-list)***
    AS *[set-type]* ***FIXPOINT***
    OF *table-name [(column-list)]*
    BY SELECT select-list
    FROM table-reference, ***view-reference***
    where-clause;

attributed-column-list ::=
    column-name [***INC*** | ***DEC*** *] [,* ***attributed-column-list*** *]*

set-type ::=
    ALL | *DISTINCT*

Fig. 1: Syntax Extension

2 Considering Architectural Alternatives

The main design variants we investigated have been to build an entirely new system completely from scratch, to integrate the new functionality into an existing system, and to construct an add-on or a frontend to an operational system. We will weigh these alternatives against each other in the sequel.

In deciding on the architectural alternative to be pursued for the proposed extensions, we considered technical as well as economic aspects. The reasons for considering technical arguments need no further explanation. The economic aspects have been considered in spite of us being located at a university institute. Since our research is mainly sponsored by governement money, we considered it important that its results would be at least in principle exploitable by some local software producer or software house without placing undue risks on the developer of customer of such a system.

2.1 Build from Scratch

The design and implementation of a new DBMS, which supports the concept of recursive views would not only be challenging, but would also offer a wealth of advantages:

* No restrictions from existing systems would have to be taken into account.
* The whole system could be constructed with special considerations to the deductive component and its implications.
* The recursive views would be deeply integrated into the DBMS (Fig. 2).
* The highest degree of optimization and, hence, highest performance, would be possible.
* One single interface for tools and application programs could be defined and the tools provided could support the complete language.

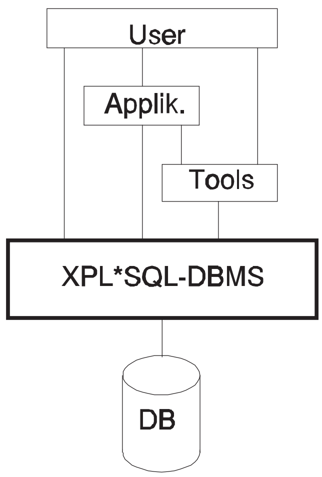

Fig. 2: Build Totally New System

The main drawbacks of this approach are the extremely high costs and the long development time that would be needed to build a DBMS totally from scratch. A great deal of the effort would be used for the design and implementation of functional aspects, which would have been only of subordinate interest in the given context. These aspects have been particularly important in our design considerations. Not only, that we didn't feel in a position to acquire the ressources for a full fledged development of an operational knowledge-base management system which would show all properties of a modern database system. We have even been sceptical about our own greediness, which might arise from good ideas in several directions off the mainstream line of thought, endangering the project to result in a never ending venture.

Besides these aspects, several aspects which might stem from the particular economic context (small country with moderate DP-industry only) in which our university is placed were considered. There is no large scale international vendor of data base systems around. Hence, the acceptance of a system based on a full integration of the database and knowledge-base aspects of the system with managers responsible for the applications to be supported by this system would have to be projected as being very low. The risk, that the developer of such a huge system might not survive would probably be too high for a responsible DP-manager.

Further, the evolutionary idea behind the construct and the language extension would be reduced to the appearance of such a system to the user (investment in training and education), since changing the vendor of one's DBMS would rather have the flair of a revolution than that of a smooth change in most of the cases.

2.2 Extending an Existing System

The internal extension of an existing system, which is well established in the market, has a much higher degree of potential for success. In contrast with the development of a totally new system, this approach poses major restrictions on design decisions, because of the high amount of investments in the basic SQL-DBMS, which must be protected. Yet it is possible to construct and present a uniform interface for users, application programs and tools. The integration of recursive views into the system and the supporting tools could be quite strong (Figure 3).

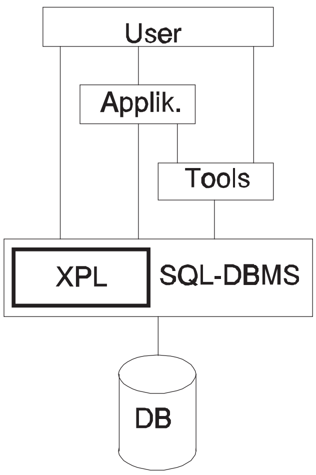

Fig. 3: Embedded Development

The extension based development would allow for moderate costs. It would also have a much higher acceptance in the market, because it would not look like a major change in the computing environment. The impact of such a system could be compared to that of a new release of a DBMS, just incorporating some (very nice) new features. However, one has to see very clearly, that such an argument would be deceiving, since the coupling between the extensions and the base-DBMS would have to be so tight, that with most modifications (new versions) of the base DBMS, a new version of the XPL-extension would also have to be supplied. This, however, would also require not only the adequate economic resources but also very intimate contact between the developer of the DBMS and the developer of the expert system extensions.

The main disadvantage of this kind of extension is that the developper of the extension must have full access to all internals (source and documentation) of an existing DBMS, and that one would have to constantly adapt the extension to the new releases of the database system itself, which usually would mean that if the developer of the extensions is not also the developer of the base system itself, he would be heavily dependend on him.

2.3 Add-on to some Existing System

This alternative form of enhancement of a DBMS is implemented in the same way as every other application program (Figure 4). Therefore, (virtually) no knowledge of the underlying DBMS internals is required.

This variant has a lot of disadvantages, if seen from a solely technical point of view. The uniform interface to other application programs and the possibility to make use of the language extension in the tools supplied with the DBMS must be given up. Further, the user has to make right from the beginning a choice, whether working with XPL or with pure SQL is needed. An awkward consequence of this choice would be that in cases, where recursive views and base views have to be used concurrently, the results of the recursive views would need to be materialized and explicitly transfered into the "ordinary" database management system, or the add-on has to be powerfull enough to process also data contained in the conventional data base of stored facts. This later option would require however full SQL capabilities and, hence, would lead us to the fourth option.

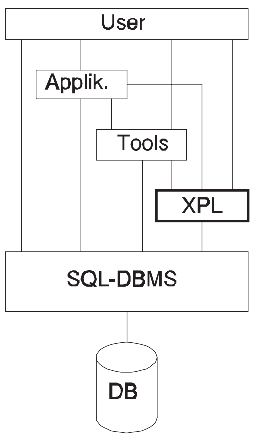

Fig. 4: Add-On to Existing System

2.4 Frontend to an Existing System

The merits of this option become directly visible, when considering the shortcommings of the adds-on alternative. Here, we do not consider the extension to be just an add-on where the clients (user, application programs and tools) have to switch between the base system and the enhancement. We rather assume it to be a real front end, allowing the clients to access the system in a completely transparent way (Figure 5).

The advantage of this solution would be - like with the previous case - that it could be implemented and maintained with comparatively moderate effort. Further, the interfaces to both, the data base management system it utilizes underneath, as well as to applications and tools would be clear cut. Therefore, no severe dependence between the developer of the DBMS and the developer of the XPL-extension would come up. Hence, even in the economic and institutional environment in which this development had to be undertaken (and for which it had been targeted), this approach seemed feasible.

Of course, there is also a price to be paid for such an architectural decision: Any SQL statement needs to be first analyzed by the XPL system and in case it is an "ordinary" SQL statement, the same analysis has to be repeated within the DBMS itself. Given the predominant structure of SQL-statements, this overhead would be marginal though. Hence, performance surely will be suboptimal due to the partly duplicated execution of operations and due to the coarse tuning of the frontend with respect to internals of the base system. Additionally, main components of the SQL-DBMS must be reimplemented (in a simplified form) in the frontend itself.

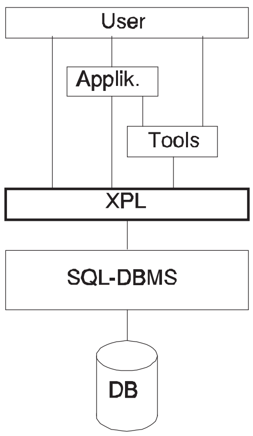

Fig. 5: Frontend to existing System

Despite the shallow integration of the frontend, it will not be completely independent from the SQL-DBMS and it will also not be portable per se, since the (highly implementation specific) catalog of the underlying system must be accessed.

From a broader perspective, however, this model doesn't look so bad as stated above, especially if one considers the possibility to market it as a special "preprocessor". This poses absolutely no hidden risk for potential customers. They can continue to use their existing DBMS, existing applications are totally unaware of the extended functionality whereas new applications can make instant use of the frontend. Since the development costs for the frontend itself can be held at a relatively low level, it would also be affordable.

This model also allows for real third-party development of the system in contrast with the internal extension of an existing system. Besides the fact that the specifications of a DBMS's external interfaces are publicly available, they also tend to be relatively stable, as compared to internal interfaces. Further, the evolutionary risk is reduced by the fact that new releases of systems are usually upwards compatible. Hence, even if the developer of the frontend cannot keep pace with the developer of the main system, the detrimental effects on the applications will be limited.

3 Architecture of the Prototype Actually Implemented

In this section, we sketch the architecture and the components of the implemented prototype, which is a frontend to an existing system (Figure 6). This decision is based on several reasons. First, we had no access to all internal information of an existing DBMS which would be necessary to extend it. Second, we had no intention to put much effort into components which are not in the center of our interest. Further, we didn't feel in the position to develop YADE (yet another database environment) and to become another DBMS vendor.

The aim of the prototype was to provide an extended SQL-based command interface, which allows one to define and query recursive views in addition to the functionality of standard SQL, and which can be used for further study.

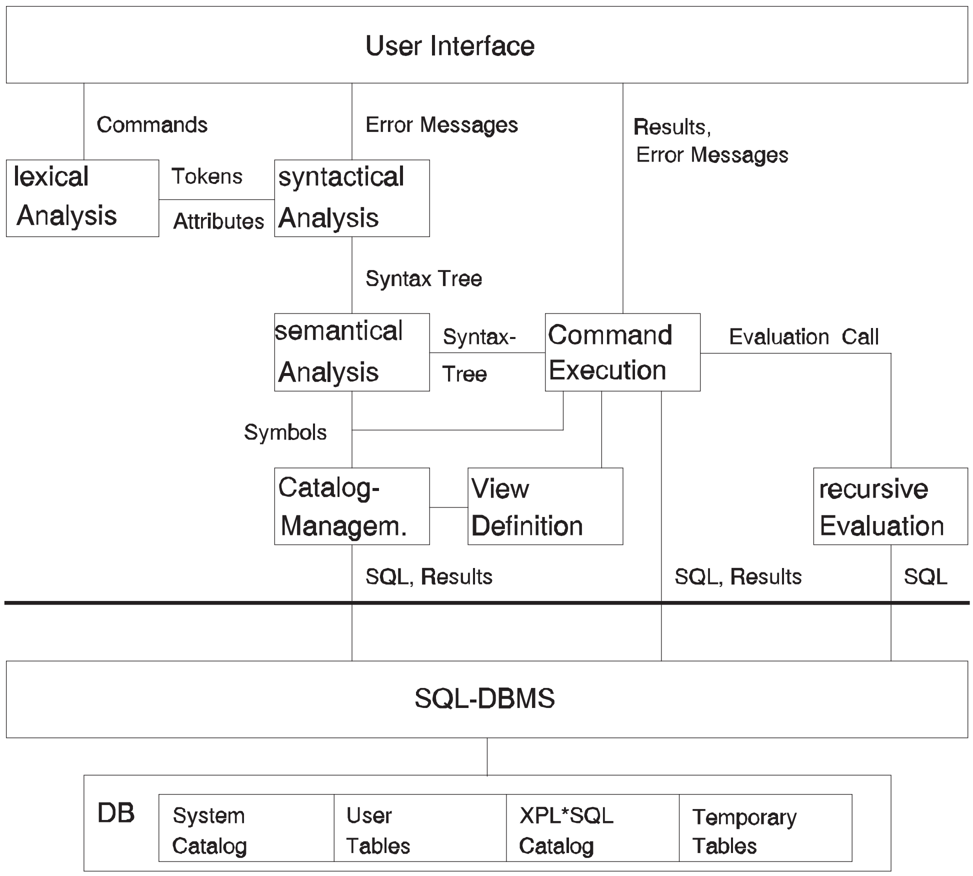

Fig. 6: Architectural Overview

The user interface component consists of a very simple line editor, which can be used to enter the extended data definition and data manipulation commands, and a rather rudimentary formatting capability for query results. Error messages are also displayed through these components. The user interface is solely character based.

All SQL commands coming from the user interface are fed into a lexical analyzer which transforms the commands from the textual form into an attributed stream of tokens.

This stream of attributes and tokens is the input for the parser. This component analyzes the stream for its syntactical correctness and constructs a syntax tree representing the structure of the command.

The semantic analysis component processes the syntax tree and extends it with new attributes. Here, not only name resolution of database objects (tables and attributes) by means of queries performed by a catalog component is carried out, but also the semantic correctness of the command is checked (at least to a certain degree).

The command executor decomposes the (possibly complex) command into smaller units, which can be executed in isolation from other units. Each unit is classified, whether it references a recursive view or just makes use of standard tables and views. If recursive views are referenced, termination and efficiency become key issues. To allow for the broadest set of safe applications [Eder90a], we check what expressions can be propagated into the computation. What is to be propagated is determined in a special part of the command executor. The thus rearranged statements are then ready for recursive evaluation. The results of this evaluation are stored in temporary tables, which are maintained by the base DBMS.

Knowing the temporary tables just computed, the command executor reconstructs an SQL-statement from the syntax tree. This SQL-command may not only be just a part of the initial SQL-command, it may also differ from it. This difference is due to the fact, that the names of the recursive views have to be substituted by the names of the temporary tables which contain the evaluation results of the recursive views referenced. The modified statement will be evaluated directly by the SQL-DBMS. Results and error messages are sent to the user interface component.

Note, that up to this point in the analysis process, all SQL-statements need to be analyzed, regardless of whether they do define or reference recursive views or not. The user does not need to switch between two different systems, and the extension is totally transparent to him.

The catalog management component updates the symbol table, based on the information contained in the system catalog of the underlying SQL-DBMS as well as in a special catalog which is used solely for the storage and retrieval of the definitions of recursive views and their corresponding attributes.

The view definition component computes the attribute dependency graph [Eder90a], which is used to classify the attributes of the view. This classification information together with the view definition is stored in the special catalog tables.

The recursive evaluator implements the algorithms to compute the results of recursive views [Eder90a, Eder90b]. It uses information from the special extendend catalog (XPL*SQL-catalog) and those constraints of the query at hand which can be propagated. The schema information concerning the relevant temporary tables is passed as a parameter to the recursive evaluator.

4 State of the Project

Currently, the implementation of a first version of the prototype is finished. It builds on the ALLBASE DBMS, running under HP-UX. It allows to interactively define recursive views and to query a database including tables, regular views and recursive views. Its actual design and implementation took about six person month.

As extensions, we forsee that the prototype could be extended to offer a programming interface allowing application programs to use the enhanced abilities of the system. A lot more of semantic checks could be added and performed in the frontend itself. This would allow for the detection of a large number of errors early in the interpretation process; errors could thus be catched before a lot of time is consumed by the evaluation of recursive views. This computation could be enhanced further by incorporating the propagation of additional kinds of restrictions into the evaluation process.

Further work will include adapting the frontend to other DBMSs and to integrate further extensions, namely extreme-value selections and aggregates. There are also plans to make use of the enhanced functionality in the context of a software engineering environment, which demands the ability to define and to use recursive views.

5 Assessment

The choosen architectural variant was adequate and allowed us to concentrate mostly on the new and specific aspects of the system without forcing us to deal with lots of internals of existing DBMS or tons of (unavailable) documentation. It was possible to demonstrate major aspects of the concepts reported in [Eder90a, Eder90b] and to substancially increase the expressive power of a relational DBMS with a rather limited effort.

We conclude that this architectural variant may be well suited when development takes place under the assumption of a third party producer with limited resources. It poses few risks, because it guarantees the highest possible independence from the vendor of the basic DBMS, and promises rather short development time with moderate cost.

Appendix:

Example of General Transitive Closure

Consider a relation *direct* with the following schema

direct(from, to, km, mins, hops)

where each of its tuples represent a direct flight which starts in city *from* and is destined to city *to*. The distance and duration of the flights are recorded in columns *km* and *mins*. The attribute *hops* contains the number of intermediate landings, which is zero in all tuples of relation *direct*, since we are considering direct flights only.

The following definition of a recursive view computes all possible flight connections between all pairs of cities, summing up distance, durations and number of hops:

```
CREATE VIEW connection (from, to, km INC, mins INC, hops INC)
AS FIXPOINT OF direct
BY SELECT d.from, c.to, d.km + c.km, d.mins + c.mins, c.hops + 1
FROM direct d, connection c
WHERE d.to = c.from;
```

This view can be used as a query target like every other table or conventional view (with some minimal restrictions, see [Eder90a]). A more complex example of an application of recursive views in the context of CPM-charts can be found in [Dobr91].

References

[Aho79] A. Aho, J. Ullmann: "Universality of Data Retrieval Languages", ACM Symp. on Principles of Programming Languages, 1979, pp. 110-120

[Boud92] N. Boudriga, A. Mili, R. Mittermeir: "Semantic Based Software Retrieval to Support Rapid Prototyping", Structured Programming, Vol. 13, No. 3, 1992

[Brod 86] Brodie M.L., Mylopoulos J.: "On Knowledge Base Management Systems", Springer Verlag, 1986

[Date87] C.J. Date: "A Guide to the SQL Standard", Addison-Wesley, Reading, 1987

[Dobr91] M. Dobrovnik: "IXPL*SQL. Erweiterung der Abfragesprache SQL um rekursive Views", Diplomarbeit, Institut für Informatik, Universität Klagenfurt, Klagenfurt, 1991

[Eder90a] J. Eder: "Extending SQL with General Transitive Closure and Extreme Value Selections", IEEE Transactions on Knowledge ans Data Engineering, Vol. 2, No. 4, Dec. 1990, pp. 381-390

[Eder90b] J. Eder: "General Transitive Closure of Relations containing Duplicates", Information Systems, Vol. 15, No.3, 1990, pp. 335-347

[Gall81] Gallaire H., Minker J., Nicolas J.-M.(eds): "Advances in Database Theory", Plenum Press, 1982

[Gall84] Gallaire H., Minker J., Nicolas J.-M.: "Logic and Databases: A Deductive Approach", ACM Computing Surveys, Vol. 16/2, June 1984, pp. 153 - 185.

[Kers86] Kerschberg L. (ed).: "Expert Database Systems", Benjamin/Cummings, 1986

[Mitt88] R.T. Mittermeir, J. Eder: "XPL*SQL. Research on new AI-Languages", Proc. 6th European Oracle User's group conference, Paris, April 1988

[SQL86] Database Language SQL, Document ANSI X3.135-1986

[Wied 86] Wiederhold G.: "Knowledge and Database Management", IEEE Software, Vol. 1/1, Jan. 1984, pp. 63 - 73