Optimization and Evaluation of Database Queries Including Embedded Interpolation Procedures

Leonore Neugebauer *
IPVR, University of Stuttgart
neugebauer@informatik.uni-stuttgart.de

Abstract

Within this paper the incorporation of interpolation procedures into a database management system is described. The interpolation functions are used to calculate intermediate values of sequences of measurements. The consequences of this class of embedded procedures for query evaluation and optimization are described. Starting with a user-friendly syntax, the interpolation procedures are decomposed into more refined operations on database objects. The impact of these operations and their predefined sequence on query evaluation and optimization is discussed. Promising evaluation strategies are emphasized.

1. Introduction

In recent years at many places demands arose for a more powerful version of the SQL [ANSI86] database language that includes embedded procedures [StAH87] and more orthogonality [Date83]. Database support is required for an increasing number of application types which current database management systems (DBMS) were not designed for. This gave rise to the development of several new DBMSs that were designed to be extensible in several aspects. Some of the best known of these systems are Starburst [HCLM90], POSTGRES [StRH90], EXODUS [CDFG86], DASDBS [SPSW90], and GENESIS [BBGS88].

Our project is mainly concerned with measurement data, i.e., data that consists of samples taken from continuously running (natural) processes. Examples are outside temperature, groundwater levels, or growth of plants. To facilitate the management of this frequently large amount of data, it can be stored within relational DBMSs. But in many cases relational operations cannot be applied directly on this data. If there are any samples missing, aggregation functions may not be applied directly; comparison by relational join is not possible due to various scales, measuring distances, or coordinates. The embedding of interpolation procedures that calculate arbitrary intermediate values offers a solution for many of these enumerated problems.

Unfortunately, the above mentioned extensible DBMSs are still in the prototyping phase and not yet widely available. Long-term maintenance cannot be guaranteed, and the systems are not ready for use by non-computer-science practitioners. Some systems, like EXODUS, explicitly require database specialists to tailor them to the requirements of the actual application.

We therefore decided to build the extensions required by our project on top of a commercially available DBMS. We extended commercial INGRES [INGR89], but any other relational DBMS that offers standard SQL, such as ORACLE, INFORMIX [Rodg90], or Sybase [Hoew86] could be easily adapted.

The main objective of our extensions is to enable the practitioner working with measurement data to query the value of a measurement process at arbitrary points. This is realized by means of interpolation procedures that are implemented in a high level programming language (C) and integrated into the query part of the DBMS. We provide users with DBMS and language extensions that exactly meet their requirements to query missing values in the original measurements, to compute equidistant sequences of measurements, and to compare different sequences using interpolation methods. These extensions are easy to apply by non-database specialists, but they do not meet the SQL standard.

Section 2 describes these extensions and shows how they can be transformed into extensions closer to standard SQL statements. The SQL-close extensions form an intermediate language more appropriate for query evaluation. These extensions consist of two types of user-defined procedures and a nesting capability. Specifically, aggregate functions, table functions that produce tables with additional attributes, and the nesting of SELECT ... FROM ... WHERE within the FROM clause, also known as table expressions, provide more orthogonality for SQL [Date83].

Section 3 derives some basic consequences of these extensions for query evaluation. The fourth section discusses methods of query optimization for queries containing these extensions. Section 5 compares our system to related work and section 6 draws some conclusions.

*) The work of the author was supported by the PWAB Project of Baden-Württemberg under grant no. PW 87 045

2. Extension of a Relational DBMS by Interpolation Procedures

When processing sequences of measurements and values reported by measuring instruments the following features are required and useful:

(1) table-like data structures for storing samples,

(2) aggregation of these values, e.g., for statistics,

(3) interpolation of missing values from surrounding samples using known methods,

(4) interpolation of equidistant sequences of intermediate values as input for models and simulation programs,

(5) comparison of different sequences of measurements.

Requirements (1) and (2) are fulfilled by conventional relational DBMSs, therefore these systems are used to store and handle measurement data. To fulfill (3) to (5), extensions to these DBMSs are necessary.

2.1 The First Approach for Sequences of Values

To motivate the extensions that will be introduced in the remainder of this section, a brief description of the typical properties of measurement values is given. For more details see [Neug89]. The most important property of measurement values is their continuity with regard to the basis against which they are recorded. A set of measurements consists of samples of a continuously running process, mainly continuous with regard to time and/or space, but possibly other dimensions such as temperature or pressure. Mathematically, the process may be considered as a continuous function

$$f_c(b_1, ..., b_n) \rightarrow m$$

$b_1, ..., b_n$ base parameters, m measured value.

The function f_c yields a value m for any possible combination of base values within their valid domains. In the database some combinations of values of the b_i are stored together with their corresponding m. Obviously, in a relational DBMS, these measurements are stored in relations with the base parameters and the measured parameter as attributes. In the following, such a relation will be called a *measurement relation*; the attributes for base parameters will be called *base attributes*, and the attribute containing measurement values of the continuous process is called the *continuous attribute*. In every measurement relation the base attributes form the key or part of the key. A measurement relation may hold several measurement attributes, all sharing the same basis. It may also hold further attributes, e.g., a description of the accuracy of the measurement value. The base attributes need not be in the relation of the measurement attributes. They can be connected via secondary keys.

Example 1:
A weather station records the air temperature every hour at an arbitrary time. The air temperature is the continuous attribute and the time is the base attribute. Another attribute contains the quality of the reported value. Fig. 2.1 shows the measured values over time.

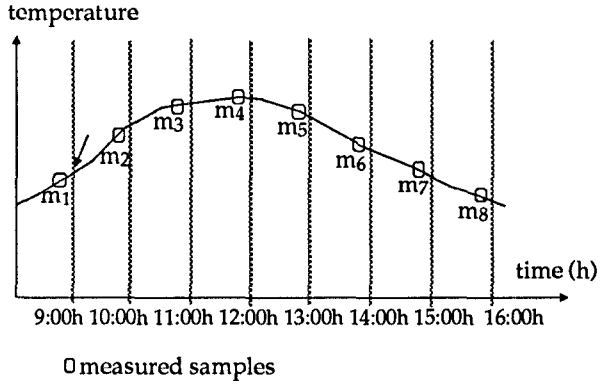

Fig. 2.1: Interpolation example

The dotted lines denote full hours, when the temperature values are required. An appropriate interpolation function would calculate a weighted mean value from the two surrounding values. If the temperature at 9:00 is queried, the weighted mean of the values m_1 and m_2 is computed. If a sequence with hourly temperatures is required, the mean value of the two surrounding samples m_i and m_{i+1} is computed for each hour. □

If measurement values are requested through base attribute values that are not explicitly stored in the database, the function f_c is evaluated for the given base value combination. Usually, this is done by deriving the required values from the surrounding samples, i.e., they are interpolated. If a measurement parameter has a many-dimensional basis, interpolation may refer to just one or a small number of base parameters. These parameters are called *varying base attributes* while the rest are called *fixed base attributes*. If the tuples in example 1 hold the coordinates of weather stations, and interpolation is done over time, then time is the varying base attribute and the coordinates are fixed base attributes.

Having these facts and user requirements in mind, the first approach to overcoming the deficiencies (from the measurement data processing point of view) of a standard relational DBMS was the following [Neug89]:

- The affected relations were marked as measurement relations, consisting of continuous attributes, (references to) base attributes, and possibly further attributes. This was done by means of additional system relations.
- Available user-supplied interpolation procedures (e.g., C functions using embedded SQL) were declared to the DBMS and their names and parameters were stored in a further system relation. This declaration specified which continuous attributes the interpolation method applied to and which base attributes were being varied.
- Syntax extensions were defined allowing interpolation to be incorporated into SQL queries (see below).
- An SQL preprocessor was implemented for interactive SQL. The preprocessor includes a new user interface behaving like the old one [Bosc90], a parser for syntax checking within the extensions [Riet90], query evaluation and optimization components that call and control the interpolation and reduce the queries to standard SQL

```
SELECT   * | [ALL | DISTINCT] [<result_column> =] <expression> {, [<result_column> =] <expression>}
FROM     <table> [<corr_name>] |
         <table> [<corr_name>] | BY METHOD <method> (<cont_attr>)                    (1)
                               ENTRY (<base_attr> = <value> {, <base_attr> = <value>}) (2)
                               STEP  (<distance> {,<distance>} )}                   (3)
         { <table> [<corr_name>] |
         [<table> [<corr_name>]] BY METHOD <method (<cont_attr>)
         ENTRY (<base_attr> = <value> {, <base_attr> = <value>})
         STEP  (<distance> {, <distance>} )} }
[WHERE <search_condition>]
[GROUP BY <attr> {, <attr>} [HAVING <search_condition>]]
[ORDER BY <result_attr> [ASC | DESC] {, <result_attr> [ASC | DESC]}];
```

Fig. 2.2: Syntax of queries extended by interpolation

```
CREATE TABLE groundwater_levels
        (x_coord      float4   BASE ATTRIBUTE,
         y_coord      float4   BASE ATTRIBUTE,
         meas_date    date     BASE ATTRIBUTE,
         meas_value   float4   CONTINUOUS  ATTRIBUTE,
         quality      char(1),
         well_id      vchar(8));
```

Fig. 2.3: Syntax of declarations of measurement relations

```
DECLARE METHOD level_interpol
        ON TABLE groundwater_levels
        CONTINUOUS ATTRIBUTE meas_value
        VARYING BASE ATTRIBUTES x_coord, y_coord;
```

Fig. 2.4: Example of the declaration of an interpolation method to the DBMS

```
SELECT   x_coord, y_coord, meas_value
FROM     ground_water_levels BY METHOD level_interpol (meas_value)
         ENTRY   (x_coord = 50.0, y_coord = 225.0)
         STEP    (0, 10.0)
WHERE    meas_date   = '29/03/90'   AND
         quality     = 'g'          AND
         y_coord     < 350.0;
```

Fig. 2.5: Example query

queries that are evaluated by the relational DBMS [Kaja90]. Interpolated values are stored in temporary relations that replace the original relations in the reduced standard queries, when they are passed to the DBMS.

The syntax of the SQL extensions is shown in fig. 2.2. (notation similar to that of [INGR89]).

Clauses (1), (2), and (3) are extensions to standard SQL. Clause (1), the BY METHOD-clause specifies the method and the continuous attribute it is applied to. Clause (2), the ENTRY-clause gives the values of the varying base attributes, i.e., it specifies the point where interpolation should by done, and clause (3), the STEP-clause gives the distances for an equidistant sequence of interpolation values.

The ENTRY and STEP clauses specify an infinite ordered set. Values given in these clauses are used by the interpolation preprocessor to generate a finite subset of the specified set. The boundaries of the generated ordered set are computed by the interpolation preprocessor dependent on the stored values and the ability of the interpolation function to extrapolate (and on further restrictions in the WHERE clause).

Example 2:
The measurement relation groundwater_levels contains tuples listing well name, coordinates, and the date, value and quality of a groundwater level measurement (see fig. 2.3). The interpolation method level_interpol computes groundwater levels at arbitrary coordinates (see fig. 2.4).

Fig. 2.5 shows a query that delivers the groundwater levels along a line where the x coordinate is equal to 50.0 on a specified date using only good ('g') measurement values. □

Though apparently similar, interpolation is distinguished from the general view mechanism, because it requires the specification and actual generation of a sequence of interpolation values.

The interpolation procedures are similar to the table functions in the Starburst extensible DBMS [HFLP88], as one of the extensions to SQL offered within this system. The main difference is that Starburst table functions may be applied to any table while interpolation procedures are assigned to one or a few (measurement) relations.

2.2 A More General Approach

The extensions proposed in section 2.1 are easy to understand and to apply by inexperienced database users. Another advantage is that interpolation can only be done within relations, i.e., measurement sets, where it is meaningful and in a way that is defined. Using this special syntax, the user is always aware of the fact that (s)he is doing interpolation and the results are not the originally measured values. However, there are shortcomings to this approach:

- In contrast to conventional SQL statements, this syntax cannot be easily transformed into the relational calculus. But this formal semantic definition is one of the major advantages of SQL.
- Search conditions of different domains are mixed up. In the example above, the condition "quality = 'g'" refers to the original relation, the condition "y_coord < 350.0" restricts the interpolated new relation, and the condition "meas_date = '29/03/90'" may be applied to either relation.
- This approach cannot easily be extended to other types of embedded procedures.
- The surrounding values to be used for interpolation must be selected within the interpolation procedure. This is an advantage for the end user because (s)he need not think about this selection, but the interpolation functions cannot be implemented without using the database and some knowledge about the structure of the stored data.
- It is difficult to compute the limits within which the step values are to be produced, if no restrictions in the WHERE clause are given.

To mitigate these shortcomings, an equivalent representation close to standard SQL and relational calculus is given. Three smaller extensions to SQL are introduced that can be combined into the desired representation (see fig. 2.6 - 2.9):

(1) The nesting of SQL queries within the FROM-clause is required. This was proposed first by [Date83] as table expressions but is not yet implemented in commercially available DBMSs. An SQL query containing this extension would look like the following:

```
SELECT  A.b1, A.b2, B.b1, B.b2
FROM    R1 A, ( SELECT   b1, b2
                FROM     R2
                WHERE    ...) B
WHERE   A.b3 = B.b1  AND ...;
```

In the inner WHERE clause references to other relations in the FROM clause are not allowed, following the SQL strategy that within nesting, references are allowed to outer relations but not to 'parallel' relations. Within interpolation, this extension is required to define the relation that holds the interpolated values.

(2) To compute the interpolated value from the surrounding values, more powerful and user-definable aggregation functions are required. Besides the attribute that is to be aggregated, the function needs further, single-valued input parameters. Within interpolation, the attribute to be aggregated represents the measurement attribute, and the single-valued parameters represent the user-defined base attributes where interpolation is to be done. This could be done with a construction like the "generalizations to arbitrary procedures" proposed by [StAH87]. Database procedures as currently offered by [INGR89a] are not sufficient because they have a fixed parameter set and cannot process sets of values as aggregation functions like SUM or AVG can do. The user-defined aggregation functions proposed in [HFLP88] are closer to what is required for interpolation, especially, if they would allow additional input parameters.

(3) A third extension is required to emulate the ENTRY- and STEP-clause given in section 2.1. To do the interpolation, a cartesian product of the original relation and the sequence defined by the ENTRY- and STEP-clause is required. Therefore, a table function is defined that has the following input parameters:
   - the original table,
   - the ENTRY- and STEP-values for each varying base attribute,
   - a *stop condition* that specifies the limits within which STEP values are to be generated.

In the following, the values generated by the ENTRY-clause, STEP-clause, and stop condition will be denoted as *step values*.

It appears reasonable to allow expressions to be given that define the stop condition dependent on the actual values that the base attributes hold, but constant expressions are possible as well. The syntax to invoke this function ('ext_table') is given in fig. 2.6.

Of course the data types of the base attributes and the assigned ENTRY and STEP values must be compatible. For example 2, the call of ext_table will look like fig. 2.7. That means that measurement devices which are more than 500 units off the interpolation point are not considered during interpolation. Fig. 2.8. shows these three extension in use.

(For simplicity, the attributes retain their names in the relation resulting from ext_table. The prefixes *old* and *new* are used to divide the original (old) values of the varying base attributes from the new ones.) If an aggregation procedure weight_3 (m, x_coord, y_coord) is defined that computes the weighted mean of the three nearest measuring wells, then example 2 would be transformed to what is shown in fig. 2.9.

The qualification "meas_date = '29/03/90'" could also be given in the outer WHERE clause. The qualification "y_coord < 350.0" could be given in the inner WHERE clause if "y_coord" were replaced by "new.y_coord".

```
ext_table ( <orig_relation>, <var_att_name1, entry1, step1, ...<var_att_namen, entryn, stepn, <stop_condition> )
```

Fig. 2.6: Syntax of the table function that generates the step values

```
ext_table (  ground_water_levels, 'x_coord', 50.0, 0, 'y_coord', 225.0, 10.0,
             sqrt (exp (abs (old.x_coord - new.x_coord)) + exp (abs (old.y_coord - new.y_coord))) <= 500)
```

Fig. 2.7: Example of the table function

```
SELECT  * | ...
FROM    R1, ... Rm, (SELECT  f_agg (m {, bi, ..., bk}), b1, ..., bn
                     FROM    ext_table (<orig_relation>, <att_namei>, entryi, stepi,
                                        ..., <att_namek>, entryk, stepk, <stop_condition>)
                     WHERE   user-defined restrictions
                     GROUP BY b1, ..., new.bi, ... new.bk, ..., bn)                    (4)
WHERE   ...;
b1, ..., bn base attributes; bi, ..., bk varying base attributes
```

Fig. 2.8: Syntax of SQL-close extensions

```
SELECT  x_coord, y_coord, meas_value
FROM    (SELECT  meas_value = weight_3 (meas_value, new.x_coord, new.y_coord),
                 x_coord = new.x_coord, y_coord = new.y_coord, meas_date
         FROM    ext_table ( ground_water_levels, 'x_coord', 50.0, 0, 'y_coord', 225.0, 10.0,
                             sqrt (exp (abs (old.x_coord - new.x_coord)) +
                             exp (abs (old.y_coord - new.y_coord))) <= 500)
         WHERE   quality     = 'g'              AND
                 meas_date =  '29/03/90'
         GROUP BY new.x_coord, new.y_coord, g.meas_date)
WHERE   y_coord < 350.0;
```

Fig. 2.9: Example 2 transformed to SQL-close syntax

If no restrictions are placed on the relation between the original base attributes and the stepwise-generated base attributes, arbitrary extrapolation could be done. If no restrictions concerning the base attribute(s) are given at all, infinite extrapolation could be done. The query evaluation component must recognize an attempt at this and either reject any such query or stop the generation of step values outside the range of existing values. (In the approach presented in section 2.1 the implementation of the interpolation procedure must ensure that the algorithm halts i.e., says from a specific distance on that no more interpolation is possible.)

This approach is more flexible than the one given in section 2.1, but is harder to read and requires more knowledge and responsible handling by the user.

3. Evaluation of Queries Including Interpolation Procedures

If the extensions presented in section 2.2 are available, interpolation procedures as given in section 2.1 could be composed of new, user-written aggregate functions, calls to the table function ext_table, and embeddings of these functions into the FROM-nested subselect given in (4), fig. 2.8. If programmers were required to write all interpolation procedures following this template, a very simple preprocessor could transform the syntax of the first approach into the second approach. Therefore, the following descriptions are given relative to section 2.2. They are also valid for section 2.1.

First of all, it must be pointed out that after the interpolation is done, the remainder of the query can be evaluated in the usual way. Thus, the first approach to extending query evaluation is to minimally extend the parts of the DBMS that evaluate the functions and that handle the temporary relations holding the interpolation results. The rest of the query evaluation and optimization is left unchanged.

Following this strategy, an extension on top of the original system can be built, e.g., on top of an existing, commercially available system.

The whole interpolation evaluation can be divided into several smaller operations, each delivering intermediate results. This is a well-known technique in query evaluation. It consists of the following steps. (Steps (1) and (2) are interchangable; all steps may be combined or mixed up with evaluation steps from other parts of the query to some extend):

(1) retrieval of the measurement relation
(2) generation of the step values
(3) supplementation of the measurement relation with the step values

(4) storage of the supplemented relation

(5) retrieval of the supplemented intermediate relation

(6) evaluation of the aggregation function

(7) storage of the intermediate interpolation relation

[(8) evaluation of the 'rest' query as usual]

The query evaluation tree and the approaches to optimization of the whole process are shown in fig. 3.1. Fig 3.2 gives the query evaluation tree frf example 2.

4. Optimization of Interpolation Queries

Optimization approaches can start at two points. First, each of the suboperations mentioned in section 3 is a candidate for optimization. Because this optimization takes place inside the interpolation procedures defined in section 2.1, it will be referred to as *inner optimization* in the following. Second, a query that contains some form of interpolation should be processed in a way that considers that interpolation. This will be denoted as *outer optimization* in the remainder of this paper.

4.1 Inner Optimization

The evaluation of queries containing interpolation starts with reading the original measurement relation and generating the step values. These two are combined into the supplemented measurement relation. One way to do this is to generate the step values (or value combinations) one after the other and supplement the appropriate tuples of the original relation with these values. If the stop condition does not contain any references to the 'old' base attributes, each original tuple is supplemented with each step value combination, i.e., a kind of cartesian product. If the original relation does not fit in main memory - and usually it does not - the relation must be read as many times as valid step values are generated. Turning back to example 2, the relation would have to be read 13 times, if the fixed stop condition "y_coord < 350.0" were used. Applying the variable stop condition given in the example to the measuring well with the highest y coordinate (10, 470), the relation would have to be read 75 times. If the stop condition contains references to the original base attributes, indexes on these attributes may be used to considerably reduce the number of tuples to be read. If the tuples are read within the interpolation procedure (the approach presented in section 2.1), just the surrounding samples are read. The optimizer can consider that, because this is always true for interpolation.

In most cases it will be advantageous to read the original relation once and generate the appropriate step values for each tuple. Generation of values is much faster than reading all the tuples. Here it is profitable to read the tuples in sequence of the fixed base attributes if an appropriate index exists, because this sequence is required in the next step of query evaluation, i.e., the grouping of the supplemented tuples by all base attributes.

If the base is many-dimensional, a many-dimensional access path would be very helpful. Index structures such as isam or btree that are available in conventional DBMSs are not sufficient. Structures that are designed for spatial data seem to be more appropriate. Examples are the buddy-tree [SeKr90], the quadtree [Same89], or cell trees with additional oversize shelves [Guen90].

During this process additional qualifications of the WHERE clause could be applied. Even conditions given in the outer WHERE clause referring to the new or fixed base attributes can sometimes be applied at this point if they do not involve any references to the outer query. In example 2, the qualification "meas_date = '29/03/90'" can be evaluated while reading the relation. This reduces the size of the intermediate relation by dividing it by the number of different measurement dates that are stored in the original relation. If 10 different dates are stored, the intermediate relation will be one tenth of the its size without this restriction. The qualification in the outer WHERE clause, "y_coord < 350.0", can be evaluated as well. If the variable stop condition is given, it reduces the number of times the original relation is read from 75 to 13.

As mentioned in section 3, the storing of the intermediate supplemented relation is the next operation to be considered for optimization. Since, it is known that the next step in interpolation will be grouping by all base attributes, storing the tuples in the sequence they are to be used in afterwards is a good idea. Here, it must be considered carefully whether storing the tuples in the required sequence, or reading and sorting the unordered relation afterwards will be more expensive. A temporary index on all base attributes is another solution to this problem. There is a choice between two alternatives:

- storing the tuples within an access structure that provides for later access in the required sequence, e.g., b-tree or isam,
- storing the tuples in the sequence they appear and building an index on all base attributes.

For example 2, the following would be useful: After the restriction in the WHERE clause reduces the fixed base attribute to a single value, an index can be built on the new x and y coordinates. Certainly this is dependent on the implementation of access structures and building of indexes in the DBMS, and the number and width of tuples in the intermediate relation. The width of tuples is known, and the number of tuples can be approximately calculated from the number of tuples in the original relation, the number of step values, and the selectivity of restrictions (if statistics are available). The time the DBMS needs to build access structures can be determined in advance with benchmarks.

After these preliminaries, the grouping of the supplemented tuples and the evaluation of the aggregation function can be done very efficiently.

The storage of the result relation, another intermediate relation, should be done with the fact that it will be read during further query evaluation in mind. Using the storage structures and indexes of the original relation might be a good idea, because this relation replaces the original relation.

An important point is the following: If the query optimizer 'knows' that a query contains interpolation, it knows that these suboperations are to be applied in sequence and can optimize using this knowledge. The generation of alternative query evaluation plans can therefore be avoided, at least for this part of the query.

4.2 Outer Optimization

The principle is to reduce interpolation whenever this can be done with reasonable effort. Instead of reading one tuple, interpolation consists of reading several tuples up to a whole relation, and the evaluation of a function. Interpolation should be done as early as possible, because the query optimizer for the 'rest' query needs information about the interpolated intermediate relation. In fact, this implies a two-level optimization; first the interpolation, and then the inclusion of the results in the query as a whole. Besides, a system built on top of an existing system cannot act in any other way.

To reduce interpolation as a whole, denesting of nested queries containing interpolation in inner subqueries is required. This corresponds to the advantages a sort-merge join has over a nested-loop join in most cases. Well-known strategies for denesting apply [Kim82, GaWo87].

If the system is fully orthogonal, this arbitrary query nesting in FROM clauses does not turn into a problem, since nested table expressions have to be evaluated first anyway, and no references to parallel tables are allowed (see above).

4.3 Performance Considerations

The performance of reading the measurement relation and supplementing it with step values can be efficiently optimized. If step values are generated for each tuple, then the time needed to read the original relation is nearly independent of the number of values that are to be interpolated.

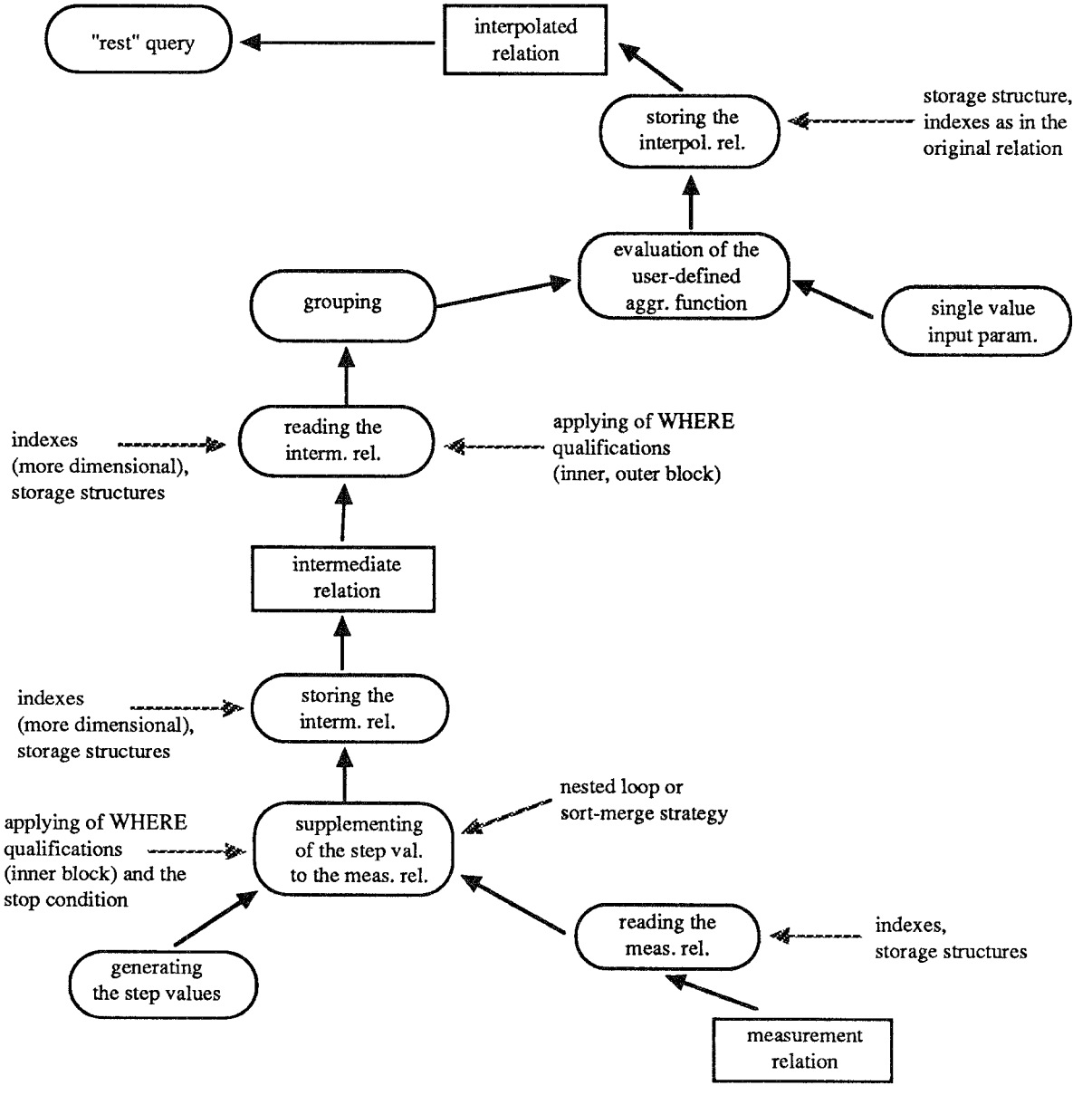

Fig. 3.1: Query evaluation tree and starting points for optimization

The retrieval time can be further reduced if, when WHERE restrictions are given, they are highly selective, and are supplied by means of indexes or storage structures. The time to read the original relation is then no longer dependent on the number of step values to be generated, but on the size of the relation alone.

If the restrictions given in the WHERE clause do not contain subqueries, they should always be evaluated as early as possible, because they have to be evaluated eventually and often they considerably reduce the number of tuples to be processed further on. Restrictions on further attributes in measurement relations should be evaluated before supplementation of the step values for three reasons:

(1) The shorter tuples can be processed faster.

(2) The intermediate relation will be smaller. If the restrictions are highly selective, the intermediate relation can be reduced by orders of magnitude.

(3) After aggregation, the values of the further attributes are no longer valid and will be removed by projection. Once the restrictions on these attributes are evaluated, they are no longer needed and the projection can be done immediately. If an intermediate relation is required, this reduces the length of the tuples and speeds up processing.

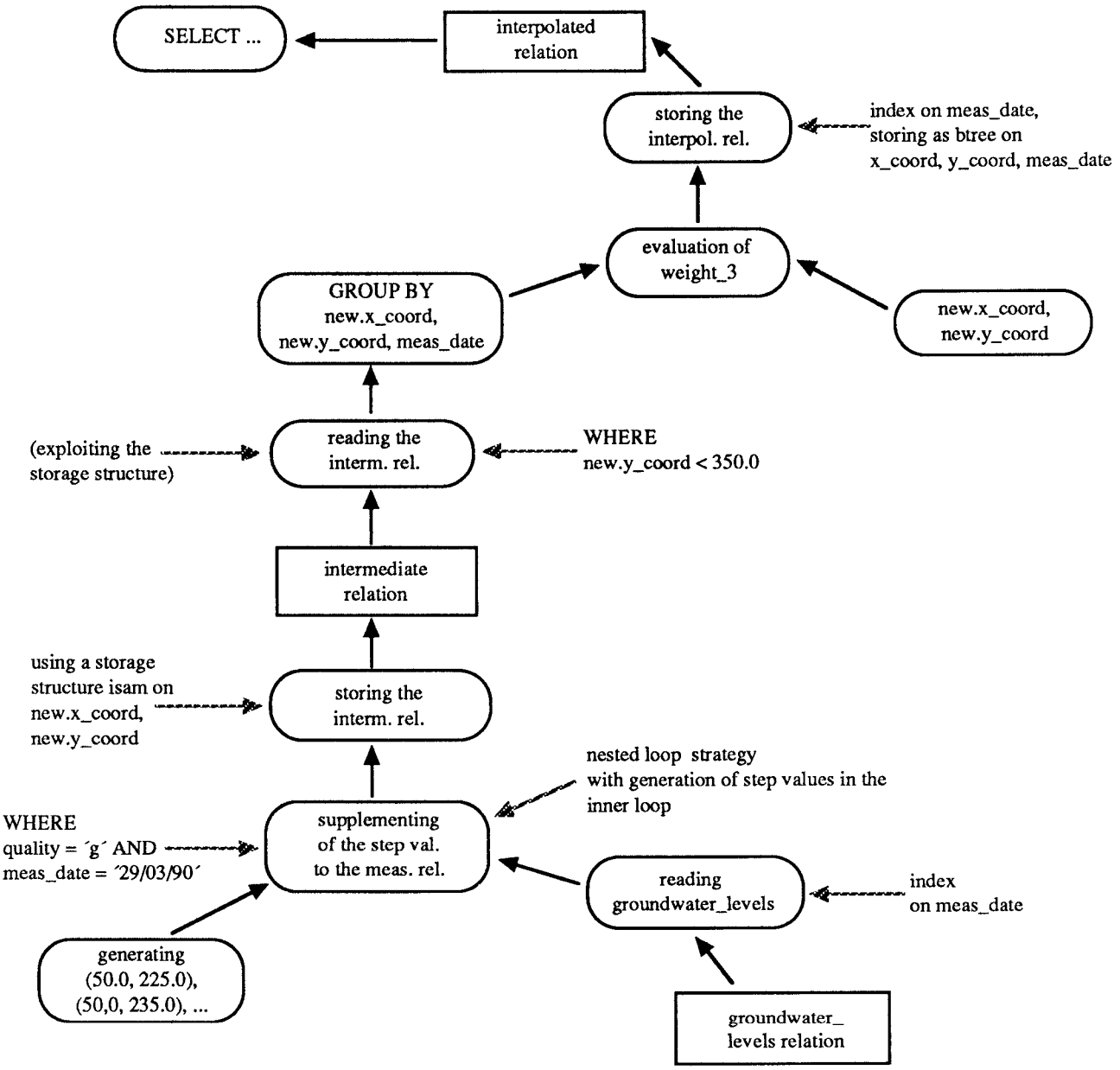

Fig. 3.2: Query evaluation tree and optimization of example 2

(1) and (2) also apply to fixed base attributes. (They do not apply to variable base attributes, however. Here, restrictions are given regarding the step values, not the original attributes.)

Within the next two steps, storage and retrieval of the intermediate relation, the performance cannot be improved that much. Whether the tuples are stored in proper order, an index is built, or the tuples are sorted after being read in arbitrary sequence, the order of magnitude is always O(n log n), if n is the number of tuples in the intermediate relation. The time required to read or store one tuple is O(log n) and there are n tuples. This matches the average time to sort n tuples, because sorting algorithms are known to be of magnitude O(n log n). The best method to use will depend on the implementation of the storage and index structures in the DBMS, the number of tuples, and their width. Similar considerations apply to the storage and retrieval of the intermediate relation containing the interpolated values.

4.4 Optimization in the Current Prototype System

The prototype system we are building [Bosc90, Riet90, Kaja90] directly evaluates the syntax given in section 2.1. At the moment, the system exploits restrictions given in WHERE clauses and uses index structures via the DBMS. The denesting of queries containing interpolation in inner subselects is currently being implemented. But the solution does have some drawbacks:

- Some of the optimizations cannot be applied. E.g., the original relation must be read several times, because interpolation is done separately for each given step value.
- Direct database access within the interpolation procedures is too troublesome and a deterrent for many practitioners who want to write their own interpolation methods. They cannot be expected to reflect on performance and to benchmark the DBMS.

This led to the approach given in section 2.2. In that approach, all the optimizations discussed could be incorporated without involving the programmer. The only disadvantage is that some intermediate relations may grow much larger. But this approach would require the SQL query language to be extended to accepting nesting in the FROM clause.

5. Related Work

The various extensible DBMS prototypes that are currently under development use different approaches for query optimization. Starburst [HFLP88] and EXODUS [Grae86] use rule-based query optimization. This approach best matches the needs for very general extensions to the query language. But this is not required just for the incorporation of interpolation. Here, fixed rules are known that can be exploited directly. This can be done by more traditional optimization methods [SACL79].

The GENESIS system [Bato86] maps the expression of its extended query language GDL to equivalent but non-optimized expressions on files and links. These so-called 'record-expressions' could be optimized in either way, traditionally or rule-based. The POSTQUEL optimizer of the POSTGRES system [StAH86] builds on the INGRES strategy. The one-variable processor had to be extended to process relation-level operators. This is done by means of extended decomposition and by the substitution of constants containing collections of QUEL commands for relation variables. The evaluation of the relation-level operators is delayed as long as possible. This is in contrast to the approach presented in this paper where the table functions are evaluated as soon as possible in order to issue reliable parameters for the optimization of the rest of the query.

6. Conclusions

The incorporation of interpolation procedures for measurement data into an existing DBMS forms a very application-driven approach to extending that DBMS. The advantage of this approach is that the extension of the DBMS by a specific kind of procedure requires a specific optimization. The fixed order of operations on database objects can be exploited. This saves the generation and assessment of different query evaluation plans. It is possible to subsequently integrate these extensions into an existing DBMS, or even to build these extensions on top of an existing system. This is a precondition to quickly building a specialized system according to specific needs, e.g., processing of environmental measurement values.

Future research could focus on topics such as whether this approach of extending an existing DBMS, including early evaluation of table expressions and of user-defined functions, is a good strategy. If so, which of the proposed optimizations are the most effective ones.

The proposed extensions can be used for many other applications that handle continuous measurement data such as engineering or chemical applications. Applications which require different extensions can be built following the same strategy.

Acknowledgements

I would like to thank Prof. Andreas Reuter for helpful discussions and continuous encouragement. Thanks to my students Thilo Jahke, Monika Bosch, Peter Rieth, and Manfred Kaja, who did most of the implementation work. Special thanks to J. David Morgenthaler for making the English more readable.

References

[ANSI86] American National Standards Institute: **Database Language SQL**; Document ANSI X.3.135-1986, 1986

[Bato86] D. S. Batory: **Extensible Cost Models and Query Optimization in GENESIS**; in: IEEE Database Engineering, Vol. 9, No. 4, December 1986, pp. 206-212

[BBGS88] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise: **GENESIS: An Extensible Database Management System**; in: IEEE Transactions on Software Engineering, Vol. 14, No. 11, November 1988, pp. 1711-1730

[Bosc90] Monika Bosch: **Einbindung von Erweiterungen in die relationale Anfragesprache SQL zur Selektion von Meßwerten über Interpolationsverfahren (Teil 1) [Incorporation of Extensions into the Relational Query Language SQL to Select Measurement Values Through Interpolation Methods (Part 1)]**; Thesis No. 872, IPVR, University of Stuttgart, 1990

[CDFG86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugenie J. Shekita, M. Muralikrishna: **The Architecture of the EXODUS Extensible DBMS: A Preliminary Report**; Computer Science Technical Report #644, May 1986, University of Wisconsin-Madison

[Date83] C. J. Date: **A Critique of the SQL Database Language**; in: ACM SIGMOD Record, Vol. 14, No. 3, November 1984, pp. 8-54

[GaWo87] Richard A. Ganski, Harry K. T. Wong: **Optimization of Nested SQL Queries Revisited**; in: ACM SIGMOD Int. Conf. on Management of Data, San Francisco, May 27-29, 1987, pp. 23-33

[Guen90] Oliver Günther: **Data Management in Environmental Information Systems**; Proc. 5. Symp. Informatik für den Umweltschutz, Vienna, Austria, September 1990, Informatik-Fachberichte 256, Springer Verlag, Berlin 1990, pp. 57-66

[Grae86] Goetz Graefe: **Software Modularization with the EXODUS Optimizer Generator**; IEEE Database Engineering, Vol.9, No. 4, December 1986, pp. 213-219

[HCLM90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, Georges Lapis, Bruce Lindsay, Hamid, Pirahesh, Michael J. Carey, Eugene Shekita: **Starburst Mid-Flight: As the Dust Clears**; in: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990, pp. 143-160

[HFLP88] L. M. Haas, J. C. Freytag, G. M. Lohman, H. Pirahesh: **Extensible Query Processing in Starburst**; Research Report RJ 6610 (63921) 12/21/88, Computer Science, IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099

[Hoew86] Linda Höwe: **Sybase Data Integrity for Online Applications**; Report, Sybase Inc., 2910 Seventh Street, Berkeley CA 94710, 1986

[INGR89] RTI INGRES: **INGRES/SQL Reference Manual**; Release 6, UNIX, April 1989, Relational Technology Inc.

[INGR89a] RTI INGRES: **INGRES/Embedded SQL User's Guide and Reference Manual**; Release 6.2, UNIX, August 1989, Relational Technology Inc.

[Kaja90] Manfred Kaja: **Auswahl und Implementierung von Optimierungsverfahren für eingebettete Prozeduren (Interpolationsroutinen) in SQL-Anfragen [Selection and Implementation of Optimization Methods to Support Embedded Procedures (Interpolation Methods) in SQL Queries]**; Diploma Thesis No. 679, IPVR, University of Stuttgart, 1990

[Kim82] Won Kim: **On Optimization an SQL-like Nested Query**; ACM ToDS, Vol. 7, No. 3, September 1982, pp. 443-469

[Neug89] Leonore Neugebauer: **Extending a Database to Support the Handling of Environmental Measurement Data**; in: Proc. Symp. on the Design and Implementation of Large Spatial Databases, Santa Barbara, Cal., July 17-18, 1989, LNCS Bd. 409, Springer-Verlag, Berlin 1990, pp. 147-165

[Riet90] Peter Rieth: **Einbindung von Erweiterungen in die relationale Anfragesprache SQL zur Selektion von Meßwerten über Interpolationsverfahren (Teil 2) [Incorporation of Extensions into the Relational Query Language SQL to Select Measurement Values Through Interpolation Methods (Part 2)]**; Thesis No. 871, IPVR, University of Stuttgart, 1990

[Rodg90] Ulka Rodgers:
UNIX Database Management Systems; Yourdon Press Computing Series, Yourdon Press, Prentice-Hall Building, Englewood Cliffs, N.J. 07632, ISBN 0-13-945593-0, 1990

[SACL79] P. Griffith Selinger, M.M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price: **Access Path Selection in a Relational Database Management System**; in: Proc. of ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass., May 30-June 1, 1979, pp. 23-34

[Same89] Hanan Samet: **Hierarchical Spatial Data Structures**; in: Proc. Symp. on the Design and Implementation of Large Spatial Databases, Santa Barbara, Cal., July 17-18, 1989, LNCS Bd. 409, Springer-Verlag, Berlin 1990, pp. 193-212

[SeKr90] Bernhard Seeger, Hans-Peter Kriegel: **The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems**; in Proc. of 16th Int. Conf. on Very Large Data Bases (VLDB), August 13-16, 1990, Brisbane, Australia, pp. 590-601

[SPSW90] Hans-Joerg Schek, Heinz-Bernhard Paul, Marc H. Scholl, Gerhard Weikum: **The DASDBS Project: Objectives, Experiences, and Future Prospects**; in: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990, pp. 25-43

[StAH87] Michael Stonebraker, Jeff Anton, Eric Hanson: **Extending a Database System with Procedures**; in: ACM ToDS, Vol. 12, No. 3, September 1987, pp. 350-376

[StRH90] Michael Stonebraker, Lawrence A. Rowe, Michael Hirohama: **The Implementation of POSTGRES**; in: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990, pp. 125-142