Query-By-Example: Operations on Piecewise Continuous Data
(Extended Abstract)

Ravi Krishnamurthy  Stephen P. Morgan  Moshe M. Zloof
IBM Thomas J. Watson Research Center, Yorktown Hts., NY 10598

Abstract

In this paper, we extend the conventional concept of a database as a set of discrete relations to include a set of piecewise continuous functions. We extend the features of Query-by-Example to operations on this piecewise continuous data. Further, we include the concept of iteration in the language, which enhances its capabilities to that of a general programming language. These extensions are accomplished without loss of the simplicity that is usually attributed to Query-by-Example; furthermore, Query-by-Example retains its table-like view of data over these new piecewise continuous functions. We present formal notions of well-formedness and correctness of Query-by-Example programs.

1. Introduction

Query languages, in general, support operations on discrete data, i.e. relations, as opposed to piecewise continuous data, i.e. piecewise continuous functions. A relation is a finite set of tuples, each representing a discrete relationship. A piecewise continuous function is a finite set of continuous mathematical functions, each representing a continuous relationship; any one such function can be viewed as a (possibly infinite) set of tuples.

Coexistence of piecewise continuous and discrete data is a natural phenomenon. For example, an experimenter (who has gathered discrete data from an experiment) frequently requires the capability of computing results from the collected data, already defined mathematical functions, and other discrete data. The coexistence of piecewise continuous and discrete data is also a common occurrence in such other applications areas as graphical databases, geographical databases, and financial and business modelling.

Traditional query languages do not allow this heterogeneity of data; besides, they are incapable of expressing the complicated computations generally required for these applications. One solution to this problem is to imbed query language statements in a high-level programming language, and, in this way, achieve coexistence of piecewise continuous and discrete data, and increased computational capability. For ad hoc queries, written perhaps by naive users, this is unreasonable.

We shall present, in this paper, extensions of Query-by-Example which allow coexistence of piecewise continuous and discrete data and which increase its expressive power. This extended Query-by-Example retains its same user-friendly interface, and in particular, its tabular view of data. In so doing, we shall introduce new constructs. Explicitly derived columns, are columns whose values are determined by a set of piecewise continuous functions imbedded in a table as data. The substitution box is extended to accommodate iterative computation. Besides the well-known base tables, (or base relations), a report type table which holds piecewise continuous data is introduced. The above extensions enable Query-by-Example to be like a general programming language, which deals not only with discrete data, but also with piecewise continuous data.

In Section 2 we review Query-by-Example and introduce its new features. In Section 3 we discuss the properties of a Query-by-Example program, and use those properties to formulate well-formedness and correctness criteria.

2. Query-by-Example and its New Features

In this section, we shall present a short review of Query-by-Example, with particular emphasis on aspects which will be of importance later on in the paper. We shall then present the concept of a piecewise continuous database, along with the new features of Query-by-Example which allow operations on piecewise continuous data, and which extend Query-by-Example's expressive power. For a more detailed review of Query-by-Example, see [Zloof75] and [Zloof77].

2.1 Query-by-Example Review

Query-by-Example allows users to operate on two-dimensional objects, including base tables, user created output tables, condition boxes, a command box, etc. A user enters various commands, such as P., I., D., U. (which mean print, insert, delete, and update, respectively), operators, such as +, >, ≠ (which have their usual mathematical meanings), constant elements (data), and example elements (variables) into object fields to define a query (program). By convention, a command is a symbol terminated with a period, an operator is a mathematical symbol, an example element is an underlined symbol, and a constant element is a quoted or unquoted string of characters.

As an example, the query of Figure 1 means: Display the names and departments of employees who earn more than $10,000.

Example elements may be used to cross reference between columns (perform joins), formulate conditions on data (perform selections and restrictions), move data from one object to an-
other (perform mappings and projections), derive new table columns, etc. Note that the same example element may achieve different relational algebraic operations based on the context in which it is used. The user has only to know about the concept of an example element and he/she can learn new applications of the same concept when necessary. One of the reasons for the simplicity of Query-by-Example is due to this property. This property is retained even in the proposed extensions. Some of these capabilities are demonstrated below.

A piecewise continuous database is a database composed of piecewise continuous relations. It can be shown trivially that the set of discrete relations is a subset of the set of piecewise continuous relations, so we incur no loss of generality if, from now on, when we speak of relations we mean piecewise continuous relations.

Now that we have defined piecewise continuous relations, let us step back and examine the question: Why are they important? Well, for one thing, they allow us to represent some relationships which were previously difficult or impossible to represent (e.g., a class of infinite relations). More importantly, they allow us to represent some relationships more naturally, within the framework of relational database theory.

2.3 New Features
In this subsection, we introduce new features of Query-by-Example which allow it to operate on piecewise continuous data, and which extend its expressive power. In particular, we introduce four new concepts into Query-by-Example: functions, derived columns, report type tables, and a substitution box. Functions and derived columns are used to define piecewise continuous relations; report type tables are a special case of piecewise continuous relations which contain no discrete data; a substitution box is used, among other things, to perform iteration on piecewise continuous data.

2.3.1 Functions and Derived Columns
In Figure 3, a column EARNINGS was constructed in the user created output table ABC by mathematical operations on data copied from the base table EMP. In base tables, the keyword FUNC allows the explicit construction of derived columns from nonderived columns. A derived column is a column whose tuple values are functionally computed from the tuple values of nonderived columns in the same table.

In Figure 4, the expression in the first row of the EMP base table explicitly defines the EARNINGS column as a function of the SAL and COMMISSION column. The asterisk preceding the column name EARNINGS specifies that this column is derived from others.

After a table is defined and its FUNC expressions and ranges are specified, data can be inserted into its nonderived columns. From a user viewpoint, the table can then be queried like any other base table. As an example, the query in Figure 5 means: Display the names and earnings of employees in the TOY department.

Query-by-Example will display the EARNINGS data after calculating it from the specified function (i.e., \( \text{EARNINGS} = \text{SAL} + \text{COMMISSION} \)), although a user will have the perception that actual EARNINGS data were stored in the EMP table.

An example element used in a function-defining expression of a derived column must be bound to nonderived columns in the same table row. Furthermore, a user cannot insert, delete, or update data in derived columns.

2.3.2 Report Type Tables
We now define a new object called the report type table, identified by the keyword REPORT preceding its table name.
report type table can be viewed as a base table which cannot contain discrete data. The difference between a report type table and a user created output table is that a report type table is a persistent object, that is it has a definition schema (including a set of mathematical functions), while a user created output has no definition schema at all. A report type table can be used as a spreadsheet, i.e. if a user enters data in nonderived columns, Query-by-Example will display corresponding computed values in derived columns.

2.3.2 Query-by-Example Programs

In this section we view a program as a set of tables; as mentioned earlier, a substitution box can be viewed as a table, and the conditions in a condition box can be mapped to corresponding example elements in the tables. We also assume that all tables are report type, since they introduce precedence constraints. The more general case where both report type tables and base tables occur in a program is discussed in [KMZ83]. In this section we present the well-formedness and correctness criteria for Query-by-Example programs. Unlike most other languages, programs in Query-by-Example do not specify any explicit order of execution. The new features pose a distinct possibility that a user may specify an ambiguous program (i.e., a program that can be interpreted in more than one way), or a meaningless program (i.e. a program that cannot be interpreted in any way). In Query-by-Example on discrete data, ordering of tables is unnecessary because of the nature of the relational operations. When we move to continuous data, two tables are implicitly ordered, as in Figure 7, if the same example element occurs in a derived column of one table and a nonderived column of the other. Consequently, a user may inadvertently construct a program implying a self-contradictory ordering requirement. To assure meaningfulness and unambiguity, well-formedness and correctness criteria are proposed, respectively.

Intuitively, a program that is well-formed assures that every nonderived column gets a value. This is achieved by requiring each nonderived column to be either a constant or a bound example element, and that no contradicting precedence constraints be implied. Thus, the well-formedness property assures that a program has a meaning.

The motivation for correctness criteria is to avoid ambiguity; i.e., the criteria should ensure the uniqueness of the meaning of a program. We define the meaning(s) of a program by equivalent sequentially ordered program(s). Unambiguity is assured by guaranteeing the uniqueness of an equivalent sequential program. This, we show, can be guaranteed by requiring from the user that every two tables that have the same example element in a derived column must be ordered by implied precedence constraints.

In this section we view a program as a set of tables; as mentioned earlier, a substitution box can be viewed as a table, and the conditions in a condition box can be mapped to corresponding example elements in the tables. We also assume that all tables are report type, since they introduce precedence constraints. The more general case where both report type tables and base tables occur in a program is discussed in [KMZ83]. In the first part of this section we define a model of a program: using this model we state, in the latter part of this section, the well-formedness and correctness criteria for a program.
3.1 Model of a Program

A program $P = \{T_i, i=1,2,...,n\}$ is viewed as a finite set of tables defined over a finite set of example elements, $E = \{e_i, i=1,2,...,k\}$, where each table includes associated conditions. Note that a program does not specify any particular ordering of execution as happens in a traditional program. Associated with each table $T_i$ are two subsets of $E$, its readset, $R(T_i)$, and writerset, $W(T_i)$. A table's readset consists of all the example elements in its nonderived columns, and its writerset consists of the example elements in the derived columns. Intuitively, each table reads the elements of its readset, carries out some computation on them and writes into the elements of the writerset.

Because of the lack of explicit ordering information on the tables, some tables may be unordered; consequently, we use parallel program schemata theory [Keller73] to define the meaning of a program on such partially ordered tables. This has been formally presented in [KMZ83]. Here we adopt a less formal approach. A parallel program schema $G$ (or a schema) for a program $P$ is a directed graph representing the precedence constraints on the tables of the program. This schema $G = (V,E)$ is constructed as follows:

$V = \text{Set of tables of the program; }$  
$E = \{(T_i,T_j) \mid W(T_i) \cap R(T_j) \neq \emptyset\}$

Intuitively, the table $T_j$ should precede $T_i$ if $T_j$ writes a value for an example element which is read by $T_i$. We use this parallel schema model to derive the properties of $P$.

3.2 The Criteria

Using the model of a program (i.e. a schema) discussed in the previous section we define well-formedness and correctness criteria for Query-by-Example programs.

3.2.1 Well-formedness Criterion

In the description of Query-by-Example, we defined a report type table to be a function that maps a set of nonderived columns to derived columns. This tacitly assumes that a proper value is given for every nonderived column. This is assured by the following criterion: A program is said to be well-formed if every entry in a nonderived column is either a constant element or a bound example element, and the program's corresponding schema is acyclic. The restriction on columns guarantees that a value is always obtainable for a column; the acyclicity ensures that no cyclic definition of the nonderived values is specified. Thus, the well-formedness criterion guarantees that every table gets a value for every nonderived column which participates in deriving another column.

3.2.2 Compile-time Correctness Criterion

One of the problems with a parallel program is that race conditions may be inadvertently specified by a user. We address this with the correctness criteria. A well-formed program is compile-time correct if the corresponding schema $G$ satisfies the following property: every pair of tables, $T_i$ and $T_j$, that conflict, are totally ordered in $G$; where $T_i$ and $T_j$ is said to conflict iff

$$[R(T_i) \cap W(T_j)] \cup [W(T_i) \cap R(T_j)] \neq \emptyset$$

This criterion guarantees that the meaning of a program is unique. A user perceives the program as a total ordering of tables (that is consistent with the partial order specified by $G$) representing a sequential program. On the other hand, there may be more than one total ordering corresponding to a partial order given in $G$. The above correctness criterion guarantees that all total ordering that corresponds to $G$ produce the same result. This has been shown in [KMZ83] and a similar result was formally proved in [Kris82]. Therefore, irrespective of which ordering is chosen as the meaning of that program by the system, the result produced is the same as that expected by the user.

This correctness criterion can be easily checked algorithmically. The algorithm must guarantee that every pair of conflicting tables must be ordered. It can be argued from the definition of $G$ that any read-write conflict between two tables will always be ordered. Thus, lack of ordering (i.e., violation of the correctness criterion), may arise only for two tables writing into the same example element. This property can be easily checked. Further, this constraint can be easily explained to a user.

3.2.3. Run-time Correctness Criterion

Let us consider an example in which tables $T_1$ and $T_2$, both write into $Z$, but for any one value of $Y$ (which is nonderived in both tables), only one of them derives a value for $Z$. This, let us say, is because the conditions on $Y$, in the two tables, are mutually exclusive. But the compile time correctness criterion will disallow this program. To enable a user to run such programs we present a new correctness criterion. A well-formed program is run-time correct if during the execution of the program all unordered conflicting tables produce values in a mutually exclusive fashion. This criterion formalizes the notion of a run-time check traditionally done by the user in his program. The execution carries out this criterion by including an appropriate check for every pair of such conflicting tables.

It is easy to see that run-time correctness criterion has the same effect as the compile-time criterion, but is less restrictive. This advantage is not without cost. Assuring compile-time correctness avoids the run-time overhead of checking, which will be significant for some programs.

4. Conclusion

We have extended the conventional concept of a database as a set of discrete relations to include a set of piecewise continuous functions. We have extended the features of Query-by-Example to operations on this piecewise continuous data. Further, we have included the concept of iteration in the language, which enhances its capabilities to that of a general programming language. All of these extensions have been done without loss of the simplicity that is usually attributed to Query-by-Example. We have also presented formal notions of well-formedness and correctness of programs in this language.

References


