MQL – a Definition and Query Language for Multidimensional Databases

Weichang Du
Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. Canada E3B 5A3
wdu@unb.ca

Abstract

This paper presents an intensional language, Multidimensional Query Language (MQL), for defining and querying multidimensional databases. Using MQL, a formal specification, or metadata, with intensional semantics can be constructed to specify multidimensional database schemas and queries in an integrated form. A multidimensional specification consists of dimension and variable declarations and definitions. Dimensions define the underlying multidimensional context space on which queries are based. Variables define data values or measures of the database which vary in the multidimensional space. A query to the database is specified by a variable definition. MQL supports dimension ranges and hierarchies by dimension definitions. It also supports incremental schema definitions and queries by declaring persistent dimensions and variables.

1 Introduction

One of the recent developments in database systems is the emergence of multidimensional databases to build data warehouses and to support decision support systems [2] [4]. It took the business world several decades to realize finally that two kinds of database systems are needed: one for business transactions (OLTP), and one for business analysis (OLAP). It has been shown that the relational model of data, which conventional relational databases are based on, is not suitable for analyzing data stored by operational database systems, from the point of view of perspectives [3]. A perspective is a factor that is associated with data values or facts. For example, the perspectives of sale values of a retail company may include dates, stores, and products. In the relational model, these perspective factors are stored as attributes with individual data values and together constitute a record of a table or relation. This is suitable for an operational database system, since the related data of each transaction can be grouped together and entered one by one into the database. However records and tables in a relational database have no clear relationships or organization in terms of perspective factors of data values. For example, all sales values from a particular date, store, or product. For analytic purposes, retrieving information from the point of view of perspectives is important, for example, to find out which store has the highest sales value for all products on a particular date.

In contrast, the multidimensional data model organizes data stored in the database according to their perspectives. It represents each perspective as a dimension. A member of a dimension is a perspective factor, such as a particular date, a store name, or a product
name. All dimensions together form a multidimensional space. Each point or cell in the space is indexed by coordinates each of which is a member of a dimension. A data value, e.g., a sales value, is stored in a cell with the index whose coordinates are the associated perspective factors of the data value. The data values at all cells as a whole constitute a measure (dimension) of the multidimensional database. Thus analyzing data values according to various perspectives in a multidimensional database can be effectively performed by dimension-wise operations (queries and manipulations), such as aggregating (consolidating) or selecting (slicing and dicing) along dimensions [5].

Currently the multidimensional data model or logical level multidimensional database is implemented at the physical level mainly by two different approaches: physical multidimensional databases (MD) or conventional relational databases (RD). A physical multidimensional database is implemented by a multidimensional (sparse) array or hypercube. This is a direct implementation of the multidimensional data model [5]. Each array dimension corresponds to a perspective dimension and is indexed by perspective factors. A data value in a cell of the hypercube is a measure. Multidimensional operations are efficient on multidimensional databases, because of their multidimensional structure and pre-calculation of aggregated data along dimensions at the database building time. There are also problems with this approach. It requires the transfer of a large amount of data from operational relational databases to the MD, because the MD is a physically separate database. It is not flexible in adding or modifying multidimensional structures incrementally, since its multidimensional structure is physically built at the database creation time. Also its space and time efficiency in terms of multidimensional operations largely depends on the effective techniques of handling multidimensional sparse arrays [8].

Logical level multidimensional databases can also be implemented physically on conventional relational databases, using the special relational schemas: star schema for plain MD and snowflake schema for MD with hierarchies [1]. In the star schema, each dimension is represented by a dimension table with attributes whose values describe properties of dimension members. Dimension tables can also have hierarchical relations, forming a snowflake schema. All data values or measures are stored in a fact table. Each attribute of the fact table is a measure. Also each record in the fact table contains references or foreign keys to members of dimension tables. Thus a multidimensional operation can be done by generating a sequence of SQL statements to manipulate the dimension tables and fact tables. The main advantage of the relational implementation is that the MD is built on the same RDBMS with the operational systems; no separate non-relational physical databases is needed. It is also flexible in terms of evolution of multidimensional structures, because the table structures of the implementation can be easily changed with SQL. However, since a multidimensional operation is implemented by a sequence of SQL statements, which are generated and executed on fly, by joining multiple tables, it is much less efficient than the direct multidimensional array implementation[7].

Multidimensional databases are designed mainly for supporting on-line analytical processing (OLAP) as part of a decision support system (DSS) [10]. Their end-users include executives, analysts and managers. End-user manipulations on MDs are usually done through an OLAP tool which provides a spreadsheet-like user interface. Through the interface, queries can be issued by selecting dimensions, defining ranges or hierarchies, and operations on measures. Multidimensional reports (usually 3D) can be generated from the query result and displayed on the interface. Using OLAP tools, the underlying implementations can be encapsulated from the users. That is, an OLAP tool can only present the user a multidimensional data model for the data which she wants to analyze, instead of showing the user
detailed operational features of the underlying implementation.

In order to understand fully and interpret the complex semantics of the multidimensional data model and operations on it, multidimensional databases do need a query and definition language with formal, implementation-independent semantics. For multidimensional databases are much more complex than spreadsheet programs which have very simple semantics. The success of relational databases largely results from the acceptance of SQL as the universal relational database query language. The declarative semantics, base on relational algebra and calculus, precisely defines the meanings of relational operations on a relational database. However, since the multidimensional data model has completely different semantics from the relational semantics of the relational data model, any variation or extension of SQL cannot be a suitable query language for multidimensional databases.

In this paper, we will introduce an intensional semantic model of the multidimensional data model and its operations. Based on the intensional semantics, we describe a declarative language MQL for defining multidimensional schemas and querying multidimensional databases. The basic idea is that we consider each data value or measure is an intension whose value varies in a multidimensional context space. A dimension in the multidimensional data model is also an intension in the context space, whose attribute values only vary in that dimension in the space. A data value at a context in the multidimensional space can be accessed by giving an attribute value for each dimension at that context.

Section 2 describes MQL language structures and its usage: multidimensional data definition and query specifications. Section 3 defines the intensional semantics of MQL constructs. Section 4 is an application example. Section 5 gives some concluding remarks.

2 Language Structure

A program in MQL, which is called a multidimensional specification, consists of five sections: dimension declaration, dimension definition, variable declaration, variable definition, and function definition.

2.1 Dimension declaration

In this section, dimension names and their attributes are declared with the following syntax.

\[\text{dimension\_variable\_name} = \text{attributes}\]
\[\quad [\text{key}] \text{type1 attribute1; }\]
\[\quad \text{type2 attribute2; }\]
\[\quad \text{end}\]

For example,

\[\text{Time = attributes}\]
\[\quad \text{key int date; }\]
\[\quad \text{int day; }\]
\[\quad \text{int month; }\]
\[\quad \text{int year; }\]
\[\quad \text{end}\]

A dimension name denotes a dimension in the program’s multidimensional context space. A dimension’s value is a set of its attribute values. Among the attributes, one attribute
can be optionally declared as a key attribute. At different contexts, a dimension may have different sets of attribute values.

The total number of dimensions in the program's context space is the total number of declared dimensions. Each dimension's value varies in its own dimension in the context space.

Each dimension in the context space is indexed *internally* by natural numbers. The internal indexes are not explicitly accessible at user program level. At an internal index, a dimension can either have defined attribute values or have a special data value *nodata*. The context at which a dimension has the first *nodata* value along the dimension defines the boundary of the dimension. The dimension has valid values only until before its boundary.

The boundary of the program's context space is defined by the boundary of each dimension. If all dimension are bounded, the program's context space is also bounded or finite.

### 2.2 Dimension definition

An undefined dimension is an input dimension, that is, its attribute values at contexts are input by the user. In terms of a multidimensional database schema, all input dimensions form the primitive multidimensional structure of the database. They are built at the database creation time. At query time, all attribute values of input dimensions should be already stored in the database.

A dimension can also be defined by another dimension using dimension operators. The definition defines the relationship of the two dimensions. A dimension definition has the form

```
newDimension = dimensionOperator(parameters) where attribute definitions end
```

Currently MQL has two built-in dimension operators: *subdim* and *nestdim*.

The binary dimension operator *subdim(d1(p))* can be used to define a sub dimension or range of a given dimension. The dimension parameter *d1* is a dimension name, and the value parameter *p* is a boolean expression whose value varies in dimension *d1*. Only attribute values of (any) dimensions and constants can be involved in *p*. The operator returns a new dimension *d2* whose values along the new dimension *d2* consist of the values of dimension *d1* at those contexts at which *p* has a true value, until *d1'*s boundary. The internal indexes of *d2*'s values is determined by the selection order of the original values of *d1*. The attribute values of *d2* can be (re)defined in the associated *where* clause. If the *where* clause is omitted, it will take copies of *d1'*s attributes. The following is an example in which we define a dimension *Tuesday* as a subdimension of the dimension *Time*.

```
Tuesday = subdim.Time(Time.day = 2)
```

The binary dimension operator *nestdim(d1(v))* can be used to define a nested dimension or hierarchy of a give dimension. It reshapes the give 1D dimension to a 2D nested dimensions. The dimension parameter *d1* is a dimension name, and the value parameter *v* is an expression of (any) ordinal data type whose value varies in dimension *d1*. Only attribute values of (any) dimensions and constants can be involved in *v*. The operator returns a new dimension *d2* whose values along the new dimension *d2* consist of *subdimensions* of *d1*. A subdimension of *d1* at each internal context *i* of *d2* consists of the values of *d1* at those contexts at which the expression *v* has the *i*th distinct value, until *d1'*s boundary. *d1'*s subdimensions at different contexts in dimension *d2* are disjoint and the union of all
the subdimensions have the same values as that of dimension $d1$. Thus the value of $d2$ is actually a 2-dimensional object varying in both $d1$ and $d2$.

The following is an example. We define dimension $Month$ as a nested dimension of the dimension $Time$ and dimension $Year$ as a nested dimension of $Month$. In other words, we define a dimension hierarchy among the $Time$, $Month$, and $Year$ dimensions.

**Dimension declaration:**

```rust
Month = attributes
    int month
    int year
end;

Year = attributes
    int year
end
```

**Dimension definition:**

```rust
Month = nestdim.Time(Time.month) where month = Time.month;
    year = Time.year end;

Year = nestdimMonth(Month.year) where year = Month.year end;
```

In the `where` clause, an attribute of the defined dimension is defined by an attribute of the defining dimension at the first context (internal index 0) in the corresponding nested subdimension.

Note that the value of $Year$ is a three dimensional object varying in $Time$, $Month$, and $Year$ dimensions. At each index $y$ of $Year$ dimension, there is a subdimension of $Month$ dimension. At each index $m$ of the $Month$ subdimension at index $y$, there is a subdimension of $Time$.

In the above, we define the hierarchy of the three dimensions with three levels. We can also define a two level hierarchy. That is, we can let $Year$ dimension be defined directly by $Time$ dimension, instead of $Month$ dimension.

```rust
Month = nestdim.Date(Date.month) where month = Time.month;
    year = Time.year end;

Year = nestdim.Date(Date.year) where year = Time.year end;
```

In this case, the $Month$ dimension and $Year$ dimension have no hierarchical relationship.

Compared with the `subdim` operator which chooses some dimension values and loses some, the `nestdim` operator never loses any dimension values. It just reshapes or rearranges the dimension values between an additional dimension and the original dimension.

### 2.3 Variable declaration

A variable has a data value or `nodata` at each context of the program’s multidimensional context space. However, depending on its dimensionality, in some dimensions a variable’s value may be constant.

A variable is declared with the data type of its data values and its dimensionality, in the following form:

```rust
data_type value_variable_name {dimension1, dimension2}
```
For example,

```plaintext
float sale {Time, Store, Product}
float monthlySale {Month, Store, Product}
```

For all input variables, i.e. those without definitions, their dimensionalities must be declared. For user-defined variables, declarations of their dimensionalities are optional. Without a declaration, the dimensionality of a user-defined variable can be obtained by the dimensionality analysis of the program.

In terms of multidimensional databases, each input variable is a primitive measure of the underlying database. Its data values are stored in the database at the database creation time. An input variable usually (but not necessary) has the dimensionality consisting of all primitive (input) dimensions. The value of a user-defined variable can be either a pre-calculated measure according to the primitive measures, or a user query result.

### 2.4 Variable definition

A variable can be defined by an expression whose value has the same data type as the variable’s. Attributes of dimensions and other variables combined with common arithmetic operations can be involved in the defining expression. All arithmetic operations are performed pointwise at each context.

MQL currently provides the following constructs and built-in operators to manipulate multidimensional data values.

The second-order operator `aggregate.d(f, e)` where $d$ is a dimension, $f$ is a binary associative arithmetic function, and $e$ is an expression with a data value type whose value may vary in dimension $d$. The `aggregate` function returns a data value which is the aggregation result by repeatedly applying $f$ to values of $e$ in dimension $d$ until $d$’s boundary. The result value is constant in dimension $d$. In other words, the `aggregate` function reduces dimension $d$ from its result’s dimensionality. For example, in the following we define a variable `totalSale` whose value is the total sale value of all dates, all stores, and all products.

**Variable declaration:**

```plaintext
float totalSale {};
```

**Variable definition:**

```plaintext
totalSale = aggregate.Time(sum,
aggregate.Store(sum,
aggregate.Product(sum, sale)));
```

The operator `at.d.a(v, e)` switches context in a dimension. Here $d$ is a dimension, $a$ is an attribute of dimension $d$, $v$ is a data value expression with the same data type as $a$’s, and $e$ is any data expression. At each context $c$ in dimension $d$, the operator returns $e$’s value at the such context along dimension $d$ at which the attribute $a$’s value of dimension $d$ is the same as $v$’s value at context $c$. If there are two or more attribute values equal to $v$’s, the first one along $d$ will be taken. If such context does not exist, it gives `nodata` value. The attribute name $a$ can be omitted. In this case, by default, the dimension’s key attribute will be used.
The operator \( at \) is a general context switching operator in a given dimension \( d \), especially when \( v \) varies in the dimension. It can be used to define data dependency between values at different contexts in the multidimensional space. For example, in the following we define a projection of 10% sale increase from the first date of the previous year.

\[
\text{projectSale} = at.\text{Time.year}(\text{Time.year-1, sale})
\]

When \( v \) is constant in dimension \( d \), the function returns the value of \( e \) in a particular context in dimension \( d \). In other words, the result is a slice of \( e \)'s multidimensional value, or \( e \) at a frozen point of dimension \( d \). The following is an example.

\[
\text{saleAt1996} = at.\text{Time.year}(1996, \text{sale})
\]

The operator \( \text{index.dimension.attribute} \) returns the attribute value of the dimension at the current context. For example, at a context \( c \), \( \text{index.} \text{Time.month} \) gives the \text{month} value of the dimension \text{Time} at \( c \).

The pointwise operator \( \text{select}(e, p) \) takes an parameter \( e \) as a general expression and \( p \) as a boolean expression. At each context, the function selects \( e \)'s value at the context if \( p \) has the true value at the context. Otherwise it will returns \text{nodata}. Considering \( e \) is an sparse multidimensional structure. The \( \text{select} \) function gives a more sparse multidimensional structure by leaving more \text{holes} (nodata) in the original structure. The following example select sale values greater than \$1000 among products and stores in each day.

\[
\text{largeSale} = \text{select(sale, sale > 1000)}
\]

Note that MQL provides two kinds of selection operations, \( \text{subdim} \) and \( \text{select} \). \( \text{subdim} \) operations build a sub-multidimensional structure by selecting dimension values in each dimension in the structure. The selection is determined only by dimension attribute values but not by data values. However, data values also can be selected using \( \text{subdim} \) operations (which we will give below). In this case, all data values within the sub-multidimensional structure are selected, and their internal indexes may be changed according to the new structure, but corresponding dimensional attributes are unchanged. In contrast, we can use the \( \text{select} \) operator to select data values according to other data values and/or dimension attribute values, without changing the positions (internal indexes) of the selected values in the original multidimensional structure. MQL does not allow whenever-like selection operations based on data values, because the operations may result in non-unique dimension values along a dimension.

A user-defined dimension, such as \( \text{Month} \), which has a nested dimensionality, can also be used as a dimension converting operator in a variable definition. The operator has the form \( \text{dimensionName}(e) \), where \( e \) is a general expression whose value usually (but not necessary) varies in the dimension which is the nested subdimension of \( \text{dimensionName} \). For example, \( \text{Month(sale)} \) where \( \text{sale} \)'s value varies in \text{Time} dimension. The operator \( \text{reshapes} \) the dimensionality of \( e \) in the same way as how the nested dimension is defined. That is, the result will have nested 2D dimensionality. For example, the result of \( \text{Month(sale)} \) consists of sale values for each store and product arranged in two dimensions \( \text{Month} \) and \( \text{Time} \). If the nested dimension (e.g. \( \text{Month} \)) is already in \( e \)'s dimensionality, the operation will return \( e \) itself.

When defining a variable varying in a user-defined nested dimension, the dimension conversions are needed before other operations can be applied to the nested dimensions. In the following examples, we define the aggregations of the monthly sale values and yearly sale values from the daily sale values, using the corresponding dimension conversion operators.
Variable declaration:
  float monthlySale {Month}
  float yearlySale {Year}

Variable definition:
  monthlySale = aggregate.Time(sum, Month(sale));
  yearlySale = aggregate.Month(sum, Year(monthlySale));

A user-defined dimension as a subdimension or range of another dimension works as a
selection operator in a variable definition, as mentioned earlier. In this case, the operator
will returns the values of a given data only within the range. For example, In the following
we define the monthly sale values in 1996.

Dimension declaration:
  Month1996 = attributes key int month; int year; end
Dimension definition:
  Month1996 = subdim.Month(Month.year = 1996);
Variable declaration:
  float monthlySale1996 {Month};
Variable definition:
  monthlySale1996 = Month1996(monthSale)

2.5 Function definition

In the Function definition part, auxiliary variables and functions can be defined. Those
defined variables are considered as neither dimensions nor variables. However in their
definitions, other dimensions and variables can be referred to. Thus they may also have
dimensionalties. Auxiliary variables can be used in definitions of dimensions and variables.
Auxiliary variables cannot be declared persistent (which is described in next section). Aux-
iliary functions may have parameters with dimension and/or variable types. In definitions
of auxiliary variables and functions, common pointwise arithmetic operators as well as prim-
itive Lucid intensional operators can be used. The intensional operators work based on the
internal dimension indexes, which we will discuss in the next section.

2.6 Schema definition and Persisted dimensions and variables

A user-defined dimension or variable can be declared persistent in its declaration. All input
dimensions and variables are persistent by default.

All persistent dimensions constitute the multidimensional schema of the underlying mul-
dimensional database. When a user-defined dimension is declared persistent, it means that
the defined dimensional hierarchy or range should be built-in, instead of computed on-fly,
in the multidimensional implementation.

All input data values are primitive which should be stored in the database at the
database creation time. The values of all persistent user-defined variables should be pre-
calculated, stored permanently in the database, and available to be directly accessed to by
following queries.

2.7 Query specification

Queries to a multidimensional database must be based on the metadata stored in the
database[6]. In MQL, the metadata consists of the schema of the database including dimen-
sion hierarchies and ranges, and the meanings of pre-calculated data values. This metadata is represented in the original form of the specification. It is always available to the user at query time. A user query can be viewed as an extension to the metadata specification.

In MQL, a variable definition is a query to the multidimensional database. The multidimensional values of the variable is the result of the query. If there are multiple variable definitions, the user can indicate a subset of the variable definitions as queries, whose results will be returned to the user. If a query is declared persistent, the query result, identified by the variable name, will be stored in the database permanently whose meaning becomes part of the metadata. Therefore, in MQL both queries and building metadata are incremental.

3 Intensional Semantics

In this section, we give the intensional semantics of MQL constructs based on the original semantics of Lucid [11]. We give the correspondence between MQL and Lucid constructs, and define the intensional meanings of MQL operators using the primitive Lucid operators and functions.

In terms of Lucid semantics, a dimension \( d \) in MQL can be considered as both a declared dimension name \( dn \) and a data variable \( du \) whose value (as a record type) varies in the \( dn \) dimension. Thus, the context space of the corresponding Lucid program consists of all declared dimensions. The variable definitions of the program is divided into two categories: dimension variables and value variable. Their values are both intensions varying in the context space. Values of dimension variables are defined by other dimension variables, and values of value variables are defined by other dimension variables and/or value variables. The dimensional information \( v \) associated with a data value at context \( c \) in a dimension \( d \), is given by \( d \)'s value at the context whose \( d \) coordinate is the same as \( c \)'s. Equivalently, since \( d \)'s value is constant to all dimensions except \( d \), \( v \) is the value of \( d \) at the same context \( c \).

The intensional semantics of the operator \texttt{subdim} can be defined using Lucid operators as follows.

\[
\texttt{subdim} \ d (\ p) = \ d \ \texttt{wvr} \ . \ d \ p
\]

The intensional semantics of the operator \texttt{nestdim} can be defined using Lucid operators as follows, where \( d0 \) is the original dimension and \( dl \) is the nexted dimension.

\[
\texttt{nestdim} \ d0 (\ v) = \ d0 \ \texttt{wvr} \ . \ d0 \ \texttt{first} \ . \ d0 \ x = x
\]
\[
\texttt{where}
\]
\[
\ x = v \ \texttt{fby} \ . \ d1 \ (x \ \texttt{wvr} \ . \ d0 \ (\texttt{first} \ . \ d0 \ x <> x));
\]
\[
\texttt{end}
\]

The semantics of the \texttt{where} clause \texttt{where d1.attr = d0.attr end} associated with \texttt{nestdim} is as follows.

\[
\texttt{d1.attr} = (\texttt{first} \ . \ d0 \ \texttt{nestdim} \ . \ d0 (\ v)) \ . \texttt{attr}
\]

The intensional semantics of the operator \texttt{aggregate} can be defined using Lucid operators as follows.

\[
\texttt{aggregate} \ . \ d (f, e) = s \ \texttt{asa} \ . \ d = \texttt{nodata}
\]
\[
\texttt{where}
\]
\[
\ s = f(e, \ \texttt{next} \ . \ d \ e) \ \texttt{fby} \ f(s, \ \texttt{next} \ . \ d \ \texttt{next} \ . \ d \ e)
\]
\[
\texttt{end}
\]
The intensional semantics of the operator at can be defined using Lucid operators as follows.

\[ \text{at.d.a}(v, e) = e \text{ asa.d} (d.a = (v0 \odot d \# d) \text{ or } e = \text{nodata}) \]

The intensional semantics of the operator index can be defined using Lucid operators as follows.

\[ \text{index.d.a} = d.a \# d \]

The intensional semantics of the operator select can be defined using Lucid operators as follows.

\[ \text{select}(e, p) = \text{if } p \text{ then } e \text{ else } \text{nodata} \text{ fi; } \]

The intensional semantics of the dimension conversion operator for a sub dimension can be defined using Lucid operators as follows.

\[ \text{subdim.d}(p, e) = e \text{ wwr.d } p \]

The intensional semantics of the dimension conversion operator for a nested dimension can be defined using Lucid operators as follows.

\[ \text{nestdim.d0}(v, e) = e \text{ wwr.d0 first.d0 } x = x \]
\[ \text{where} \]
\[ x = v \text{ fby.d1 } (x \text{ wwr.d0 } (\text{first.d0 } x \leftrightarrow x)); \]
\[ \text{end} \]

4 An Application Example

In the following, we give a complete multidimensional specification in MQL for an automobile marketer application [5].

4.1 Database schema and creation

The database consists of four primitive dimensions models and colors of automobiles, dealerships, and times. The primitive data value is the sale volume of each model and color by each dealer on each day. In the specification, the primitive dimensions and data value are specified as the input dimensions and variable.

There are three dimension hierarchies. The hierarchy built on the time dimension has three levels: date, month, and year. There are two 3-level hierarchies built on the dealership dimension. One is based on the sale organization with levels dealership, district, and region. The other is based on the distribution organization with levels dealership, distribution point, and import point. In the specification, the hierarchies are specified by the persistent user-defined dimensions as the nested dimensions of the primitive ones.

The primitive dimensions and variable and the hierarchies together constitute the schema of the database.

At the database creation time, total sale volumes at each non-primitive level of each hierarchy is pre-calculated and stored in the database. In the specification, the consolidations are specified by the persistent user-defined variables.
Dimension declaration:
  Model = attributes key string name end;
  Color = attributes key string color end;
  Time = attributes key int date; int day; int month; int year end;
  Dealership = attributes
    key string name;
    string district;
    string region
    string saleOrg;
    string distributionPoint;
    string importPoint;
    string distributionOrg;
  end;

  persistent Month = attributes key int month; int year end;
  persistent Year = attributes key int year end;

  persistent District = attributes key string name; string region: string Org end;
  persistent Region = attributes key string name; string Org end;

  persistent DistributionPoint = attributes
    key string name;
    string importPoint;
    string Org end;

  persistent ImportPoint = attributes key string name; string Org end;

Dimension definition:
  Month = nestdim.Time(Time.month) where month = Time.month; year = Time.year end;
  Year = nestdim.Month(Month.year) where year = Month.year end;

  District = nestdim.Dealership(Dealership.district);
    where
      name = Dealership.district;
      region = Dealership.region;
      org = Dealership.saleOrg;
    end;

  Region = nestdim.District(District.region)
    where
      name = District.region;
      org = District.org end;
    end;

  DistributionPoint = nestdim.Dealership(Dealership.distributionPoint);
    where
name = Dealership.distributionPoint;
importPoint = Dealership.importPoint;
org = Dealership.distributionOrg;
end;
ImportPoint = nestdim.DistributionPoint(DistributionPoint.importPoint);
where
    name = DistributionPoint.importPoint;
    org = DistributionPoint.org;
end;

Variable declaration:
    int saleVolume {Model, Color, Dealership, Time};
    persistent int monthlySale {Model, Color, Dealership, Month};
    persistent int yearlySale {Model, Color, Dealership, Year};
    persistent int districtSale {Model, Color, District, Time};
    persistent int regionSale {Model, Color, Region, Time};
    persistent int distributionSale {Model, Color, DistributionPoint, Time};
    persistent int importSale {Model, Color, ImportPoint, Time};

Variable definition:
    monthlySale = aggregate.Time(sum, Month(saleVolume));
    yearlySale = aggregate.Month(sum, Year(monthlySale));
    districtSale = aggregate.Dealership(sum, District(saleVolume));
    regionSale = aggregate.District(sum, Region(districtSale));
    distributionSale = aggregate.Dealership(sum, DistributionPoint(saleVolume));
    importSale = aggregate.DistributionPoint(sum, ImportPoint(districtSale));

4.2 Queries

1. Consolidation

A consolidation operation aggregates, in some way, a collection of data items along one or more dimensions to reduce them to a single value. In MQL consolidation operations are specified by applications of the aggregate operator, combined with other operators.

In the following MQL query, we specify a consolidation avgSatRegSale which is the average sale volume on Saturdays for each model and color at each region. In the query, we first define an auxiliary subdimension Saturday of the Time dimension, then we aggregate along the Saturday dimension to compute the aggregation, producing the result consolidation avgSatRegSale which varies in neither Time nor Saturday dimension.

Dimension declaration:
    Saturday = attributes int day end;
Dimension definition:
    Saturday = subdim.Time(Time.day = 6) where day = Time.day end;

Variable declaration:
    int avgSatRegSale {Model, Color, Region};

Variable definition:
    avgSatRegSale = aggregate.Saturday(avg, Saturdays(regionSale));

2. Slicing and dicing

A slicing operation freezes a multidimensional data object at fixed indexes in one or more dimensions. The result of the operation is constant in the frozen dimensions which is the values at the frozen points. In MQL slicing operations are specified by applications of the at operator, combined with other operators.

In the following MQL query, we specify a slice blue VanSale which is the daily sale volume of blue minivans at Chicago district. It freezes three dimensions: Color dimension at blue, Model dimension at minivan, and District dimension at Chicago.

Variable declaration:
    int blueVanSale {Time};

Variable definition:
    blueVanSale = at.Color("blue", at.Model("minivan",
                at.District("Chicago", districtSale)));

A dicing operation reduces a multidimensional data object to a smaller ranges in certain dimensions. In other words, given a multidimensional data object as a hypercube. A dicing operation produces a sub-hypercube of the given one. In MQL dicing operations are specified by defining subdimensions.

In the following MQL query, we specify a dice subSale whose dimensionality consists of subdimensions of Model, Color and Dealership. In the sub-model dimension sport coupe and mini van are selected, in the sub-color dimension blue and red are selected, and in the dealership dimension Carr and Clyde are selected.

Dimension declaration:
    Submodel = Model;
    Subdealer = Dealership;
    Subcolor = Color;

Dimension definition:
    Submodel = subdim.Model(Model.name = "sports coupe" or
                Model.name = "minivan");
    Subcolor = subdim.Color(Color.name = "blue" or
                Color.name = "red");
    Subdealer = subdim.Dealership(Dealership.name = "Carr" or
                Dealership.name = "Clyde");
Variable declaration:
    int subSale {Submodel, Subcolor, Subdealer, Time};

Variable definition:
    subSale = Submodel(Subcolor(Subdealer(saleVolume)));

3. Roll-up and Drill-down

A roll-up operation works along a dimension hierarchy. At a context in a lower-level dimension, it looks up data at the context in the dimension of the above level. The higher-level context include the lower level context as one of the values of its subdimension. The upper level data can be an aggregated data or a collection of lower-level data values corresponding to the subdimension of the higher-level context. For example, from the sale volume at a particular dealership, say Clyde, the roll-up operation can look up the total sale volume of the Chicago district, or the sale volumes of all dealers in the Chicago district.

Symmetrically, a drill-down operation can look up from a higher-level dimension context to a lower one. For example, from the sale volume in a particular district, say Chicago, the drill-down operation can look up the sale volumes of all dealers in the district.

In MQL the nested dimension construct naturally supports roll-up and drill-down operations. Using nested dimension operations, we can build a dimension hierarchy from primitive data values without losing information. That is, we build a hierarchical structure with nested dimensions using the exact same data set. In the following specification, we define the hierarchy of the sale volumes in Sale organization-Region-District-Dealership dimension hierarchy.

Variable declaration:
    int regionSaleAll {Model, Color, Time, Dealership, District, Region};
    int districtSaleAll {Model, Color, Time, Dealership, District};

Variable definition:
    districtSaleAll = District(saleVolume);
    regionSaleAll = Region(districtSaleAll);

In the following MQL query, we specify roll-up and drill-down operations along the sale organization dimension hierarchy with non-aggregate data. The aggregated data in the hierarchy has been pre-calculated and stored in the data base by the earlier specification.

Variable declaration:
    int midwestRegionSale {Model, Color, Time, District};
    int chicagoDistrictSale {Model, Color, Time, Dealership};
    int clydeDealerSale {Model, Color, Time};

    int clydeDistrictSale {Model, Color, Time, Dealership};
    int clydeRegionSale {Model, Color, Time, Dealership, District};
    int clydeDistrict {};

14
Variable definition:

// Drill-Down
midwestRegionSale = drilldown.Region("Midwest", regionSaleAll);
chicagoDistrictSale = drilldown.District("Chicago", midwestRegionSale);
clydeDealerSale = drilldown.Dealership("Clyde", chicagoDistrictSale);

// Roll-Up
clydeDistrictSale = rollup.Dealership.District
("Clyde", districtSaleAll);

clydeRegionSale = rollup.District.Region
(clydeDistrict, regionSaleAll);

clydeDistrict = first (rollup.Dealership.District
("Clyde", District));

Function definition:

drilldown.dim(key, exp) = at.dim(key, exp);

rollup.lowdim,highdim(lowKey, highExp) =
at.highdim(highKey, highExp)
where highKey = at.lowdim(lowKey, index.highdim) end;

4. Pivoting

In an OLAP system, the result of a query to the underlying multidimensional database will be output, as generated reports, on a 2D or 3D display. If the result is more than 3D, some dimensions have to be reduced by either slicing or aggregate. Also in the display, there are various ways to layout the result dimensions to the display dimensions. Rotating and displaying a same query result from different dimensional angles is called pivoting operations in OLAP. In the recently published standard API for multidimensional databases all query results must be mapped to a three dimensional cube object as the final results through the API [9].

MQL as a formal query language for multidimensional databases is not intended to specify how query results being displayed. However, to show the multidimensional expressive power of intensional programming, and as an extension to supporting OLAP operations, in the following we define the pivoting operation as a Lucid function, and show some pivoting examples in MQL.

Let the three display dimensions be $R$ for Row, $C$ for Column, and $P$ for Page. A pivot operator with three formal dimensional parameters is defined as follows, by simply using Lucid’s realign operator.

$$\text{pivot}_{x,y,z}(\text{exp}) = \text{realign}_{x,R}(\text{realign}_{y,C}(\text{realign}_{z,P}(\text{exp})))$$

Let $allDealerSale$ be the total sale of all dealers for each model and color on each day.

Variable declaration:
int allDealerSale (Model, Color, Time);

Variable definition:
allDealerSale = aggregate.Dealership(sum, saleVolume);

In the following specification, we display the 3D data value allDealerSale in all six
different dimension layouts using the pivot operation. Each of them can be considered
as a subject to the above MQL specification.

pivot.Model,Color,Time(allDealerSale);
pivot.Color,Time,Model(allDealerSale);
pivot.Time,Model,Color(allDealerSale);
pivot.Model,Time,Color(allDealerSale);
pivot.Color,Model,Time(allDealerSale);
pivot.Time,Color,Model(allDealerSale);

5 Concluding Remarks

In this paper we introduced a preliminary version of MQL. It shows the expressive power,
formalism, and simplicity of the language for multidimensional database definitions and
queries. It is because multidimensional specifications in MQL is based on intensional se-
manics which is proved best for manipulating multidimensional objects. As mentioned in
the introduction, the reason that multidimensional database systems are not as powerful
and widely accepted as relational databases is because it lacks formalism for their definitions
and queries. The best that the OLAP and MB developers can agree on so far is a standard
API to OLAP/MD engines [9]. The API has only informal operational semantics. Their
attempts to design a standard query language so far is not quite successful. For all proposed
query languages are extensions and/or variations to the relational SQL [5]. We believe that
the multidimensional data model should be orthogonal to the relational model, and a new
non-relational semantics should be defined. That is where the intensional semantics fits.

Currently we are working on a prototype Intensional OLAP (IOLAP) system based
on MQL. The system accepts incremental MQL specifications as inputs to define database
schemas, and to outputs query results and the metadata of the underlying database for
further queries. The system implementation compiles MQL specifications to the standard
OLAP/MD API code.

References

[3] Sahin, K., Multidimensional database Technology and Data Warehousing, Database
[4] Raden, N., Multidimensional Data Modeling: The Data Model for the Rest of Us, Informa-
tion Week, Jan. 1996.


