CoherentPaaS

Coherent and Rich PaaS with a Common Programming Model

ICT FP7-611068

Common Query Language and Data Model

D3.1

<March, 2014>

Document Information

Scheduled delivery	31.03.2014
Actual delivery	06.05.2014
Version	1.4
Responsible Partner	INRIA

Dissemination Level:

PU Public

Revision History

Date	Editor	Status	Version	Changes
13.01.2014	Kolev	Draft	0.1	Initial draft
03.03.2014	Kolev	Final	1.1	Submitted for review
17.03.2014	Kolev	Revised	1.2	Reflected MonetDB review
25.03.2014	Kolev	Revised	1.3	Reflected QuartetFS review
06.05.2014	Kolev	Final	1.4	Renamed MdbQL to CloudMdsQL

Contributors

Boyan Kolev, Patrick Valduriez

Internal Reviewers

MonetDB, QuartetFS

Acknowledgements

Research partially funded by EC 7th Framework Programme FP7/2007-2013 under grant agreement n° 611068.

More information

Additional information and public deliverables of CoherentPaaS can be found at: http://coherentpaas.eu

Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
CloudMdsQL	Cloud Multi-datastore Query Language
DBMS	DataBase Management System
DBPL	DataBase Programming Language
DDL	Data Definition Language
DML	Data Manipulation Language
DQL	Data Query Language
ETL	Extract, Transform, Load
FIRA	Federated Interoperable Relational Algebra
FISQL	Federated Interoperable Structured Query Language
JSON	JavaScript Object Notation
MDX	Multi-Dimensional eXpressions
NoSQL	Not only SQL
RDBMS	Relational DataBase Management System
SQL	Structured Query Language
XML	eXtensible Markup Language

Table of Contents

List of Figures

List of Tables

1. Executive Summary

The common query language CloudMdsQL is designed to be capable of querying multiple heterogeneous databases (relational and NoSQL) within a single query containing nested sub-queries. During query execution the query engine for CloudMdsQL needs to run native queries against a diverse set of data stores, and integrate the results according to the common data model.

The dominant state-of-the-art solution for integrating multiple heterogeneous data sources is the mediator/wrapper architecture with global mediated schema, providing transparent access to data sources, thus hiding data source heterogeneity and distribution. Each wrapper has to provide translations between the source data and schemas and the mediated schema, while the mediator centralizes the information provided by the wrappers in a global schema and integrates the datasets retrieved from the data stores. For the query language, there are three major requirements: to allow nested queries across different data sources, schema independence and ability to perform data-metadata transformations. [8]

CloudMdsQL sticks to the relational data model, because of its intuitive data representation, wide acceptance and ability to integrate datasets by applying joins, unions and other relational algebra operations. To be robust against schema evolution, CloudMdsQL keeps its common data model schema-less, while at the same time it is designed to ensure that all the datasets retrieved from the data sources match the common data model. The common data model supports basic relational operators (projection, selection, joins, aggregation, sorting, union, etc.). To support data and data-metadata transformations, CloudMdsQL introduces an operator which can perform transformations over intermediate relations and/or generate synthetic data by executing embedded code of a functional language, part of CloudMdsQL. The requirement of nesting queries from heterogeneous data sources implies the usage of the bind-join operator [9], which uses the data retrieved from one data source as an input to a query to another data source.

The CloudMdsQL language itself is SQL-based with the extended ability for embedding native queries to data stores and programming language constructs, necessitated mostly by the requirement for data and data-metadata transformation and by the fact that some data sources have API-based native query interface. To support such functional programmability, CloudMdsQL queries can contain constructs of the programming language Python, the choice of which is justified by its richness of data types, ease of use, richness in standard libraries and wide usage.

An important concept introduced by CloudMdsQL is the notion of “table expression”, which is generally an expression that returns a table (relation – a structure, compliant with the common data model). Table expressions are used to represent nested queries and most often address a particular data store. Three kinds of table expressions are distinguished:

- SQL table expressions, which are regular nested SELECT statements;
- Embedded blocks of Python statements that produce relations;
- Native table expressions, using a data store’s native query language.

A table expression is usually assigned a name and a signature, thus turning it into a "named table expression", which can be used in the FROM clause of the query as a regular relation. Named table expression's signature defines the names and types of the attributes of the returned relation. Thus, each CloudMdsQL query is executed in the context of a kind of ad-hoc schema, formed by all named table expressions within the query. This approach fills the gap produced by the lack of a global schema and allows the query compiler to perform semantic analysis of the query. A named table expression is usually defined as a query against a particular data store and contains references to the data store's data structures. However, the expression can also instantiate other named table expressions, defined against other data sources, thus chaining data as per the requirement for nesting queries.

Similarly to table expressions, CloudMdsQL introduces the notion of "action expressions", which are composed of statements that perform data modification operations against the corresponding data store. They are used in data manipulation CloudMdsQL statements and instantiated by an EXECUTE clause, which can be more than one for a single CloudMdsQL query. For an SQL data store an action expression contains SQL DML command, while for a NoSQL data store an action expression performs invocations to the data store's native query API. An action expression can instantiate named table expressions which gives the flexibility for a single query to retrieve data from one (or more) data store, perform transformations on it and then use it to update another data store. A single CloudMdsQL command can perform data manipulation against several data stores.

Named expressions can be parameterized, thus making table expressions behave like parameterized views and action expressions – like parameterized procedures. CloudMdsQL also allows a (parameterized) table or action expression to be given a global name and stored in a global context in order to be referenced in several CloudMdsQL queries, similarly to SQL stored procedures/functions.

2. Introduction

The common query language CloudMdsQL is designed to be capable of querying multiple heterogeneous databases (relational and NoSQL) within a single query containing nested sub-queries. During query execution the query engine for CloudMdsQL needs to run native queries against a diverse set of data stores, and integrate the results according to the common data model.

2.1. Mediator/Wrapper Architectural Model

The problem of accessing and integrating heterogeneous data sources, i.e. managed by different data management systems such as RDBMS or XML DBMS, has long been studied in the context of multi-database systems (also called federated database systems) [6] and data integration systems for the Web [3]. The typical solution is to provide a common data model and query language to transparently access data sources, thus hiding data source heterogeneity and distribution.

The dominant state-of-the-art architectural model is the mediator/wrapper architecture, where each data source has an associated wrapper.

Each wrapper:

- Exports information about the source schema and mapping functions that translate between the source data and schemas and the mediated schema;
- Transforms queries expressed in the common language into queries for the DBs;
- Transforms the queries' results in the common data model.

The mediator:

- Centralizes the information provided by the wrappers in a global schema;
- Transforms queries expressed in the common language into queries for the wrappers;
- Integrates the queries' results.

The mediator/wrapper architecture has several advantages. First, the specialized components of the architecture allow the various concerns of different kinds of users to be handled separately. Second, mediators typically specialize in a related set of data sources with "similar" data, and thus export schemas and semantics related to a particular domain. The specialization of the components leads to a flexible and extensible distributed system. In particular, it allows seamless integration of different data stored in very different data sources, ranging from full-fledged relational databases to simple files.

2.2. Common Query Language Requirements

The main requirements for a common query language (and data model) are support for nested queries, schema independence and data-metadata transformation [8]:

- Nested queries: to allow queries to be arbitrarily chained together in sequences, so the result of one query (for one data source) may be used as the input of another (for another data source).
- Schema independence: to allow the user formulate queries that are robust in front of schema evolution.
- Data-metadata transformations: to deal with heterogeneous schemas by transforming data into metadata and conversely, e.g. data into attribute or relation names, attribute names into relation names, relation names into data.

These requirements are not supported by query languages designed for centralized databases, e.g. SQL and XQuery. Therefore, federated query languages need major extensions of their centralized counterpart.

We now discuss briefly two kinds of such extensions of major interest for CoherentPaaS: relational languages and functional SQL-like languages. In [8], the authors propose an extended relational model for data and metadata integration, the Federated Relational Data Model, with a relational algebra, Federated Interoperable Relational Algebra (FIRA) and an SQL-like query language that is equivalent to FIRA, Federated Interoperable Structured Query Language (FISQL). FIRA and FISQL support the requirements discussed above, and the equivalence between FISQL and FIRA provides the basis for distributed query optimization. FISQL and FIRA appear as the best extensions of SQL-like languages for data and metadata integration. In particular, it allows nested queries. But as with SQL, it is not possible to express some complex control on how queries are nested, e.g. using programming language statements such as IF THEN ELSE, or WHILE. Note that, to express control over multiple SQL statements, SQL developers typically rely on an imperative language such as Java in the client layer or a stored procedure dialect such as PLSQL in the database layer. Another major limitation of the relational language approach is that it does not allow exploiting the full power of the local data source repositories. For instance, mapping an SQL-like query to a Graph Database query will not exploit the Graph DBMS capabilities, e.g. generating a Best First Search query.

Database programming languages (DBPLs) have been proposed to solve the infamous impedance mismatch between programming language and query language. In particular, functional DBPLs such as FAD [2] can represent all query building blocks as functions and function results can be used as input to subsequent functions, thus making it easy to deal with nested queries with complex control. The first SQL-like functional DBPL is Functional SQL [7]. More recently, FunSQL [1] has been proposed for the cloud, to allow shipping the code of an application to its data. Another popular functional DBPL is LINQ [5], whose goal is to reconcile object-oriented programming, with relations and XML. LINQ allows any .NET programming language to manipulate general query operators (as functions) with two domain-specific APIs that work over XML (XLinq) and relational data (DLinq) respectively. The operators over relational data provide a simple object-relational mapping that makes it easy to specify wrappers to the underlying RDBMS.

3. Design Considerations and Common Data Model

CloudMdsQL is an SQL-like functional language and its query engine sticks to the traditional mediator/wrapper approach; however it goes beyond the state-of-the-art in that it integrates fully-functional queries to several databases that differ significantly from each other in terms of data models and query interfaces. CloudMdsQL has the capability to exploit the full power of the local data stores by embedding calls to data stores' native query interfaces without loss of functionality. For the purpose, CloudMdsQL keeps its common data model schema-less, while at the same time it is designed to ensure that all the datasets retrieved from the local databases match the common data model.

3.1. Data Model

The proposed data model is table-based (relational), because of several reasons:

- The relational data model provides simple and intuitive data representation (tables);
- It allows different datasets to be easily integrated by applying binary relational operations like joins, unions, etc.;
- SQL is a well-known standard, familiar to users and developers with SQL APIs widely used by many tools.

The common data model is schema-less, because:

- NoSQL databases can be schema-less, which makes it almost impossible to derive a global schema;
- Mapping local to global schemas and data might limit the capability of the query engine to exploit the full power of local data stores' query interfaces.

The basic operators that can be performed over relations are exactly as they appear in the relational data model:

- Projection;
- Selection;
- Joins (incl. inner joins, outer joins, full outer joins, semi-joins, bind-joins);
- Aggregation;
- Sorting;
- Set operations: union, intersection, set difference.

In order to be capable to perform powerful data-metadata transformations, CloudMdsQL introduces another operator which can perform transformations over intermediate relations and/or generate synthetic data by executing embedded code in a functional language, which is part of CloudMdsQL.

The requirement of nesting queries from heterogeneous data sources implies the usage of the bind-join operator, which uses the data retrieved from one data source as an input to a query to another data source. [9]

3.2. Data Types

The CloudMdsQL data model supports a minimal set of data types (explained in Appendix A), enough to capture data types supported by the data models of most data stores:

- Scalar data types: integer, float, string, binary, timestamp;
- Composite data types: array, dictionary (associative array);
- Null values.

Standard operations over the above data types are also available: arithmetic operations, concatenation and substring, as well as operations for addressing elements of composite types (e.g. array[index] and dictionary['key']).

3.3. Query Language

The common query language CloudMdsQL is based on the SQL standard with the extended possibilities for defining table expressions and embedding programming language constructs.

A table expression is generally an expression that returns a table (relation – a structure, compliant with the common data model). Table expressions are used to represent nested queries. Three kinds of table expressions are distinguished:

- SQL table expressions, which are regular nested SELECT statements;
- Embedded blocks of programming language statements that produce relations;
- Native table expressions, using a data store's native query language.

A table expression is usually assigned a name and a signature in order to be used in the FROM clause of the query as a regular relation.

The programming language constructs within CloudMdsQL are generally used to:

- Define named table expressions;
- Invoke specific API methods to query NoSQL data stores;
- Convert arbitrary datasets to relations in order to comply with the common data model;
- Complement the query language with functional capabilities;
- Perform data-metadata transformations;
- Perform type conversions.

3.4. Python as Functional Extension of CloudMdsQL

To achieve functional programmability as mentioned above, we propose that CloudMdsQL queries contain embedded constructs of the programming language Python. The choice of Python is justified with the following arguments:

- It supports all data types from the common data model (including Null values);
- Many DBMSs have Python APIs (including Dex, MongoDB, MonetDB);
- It is simple, fairly well-known and easy to use;
- It is rich in standard libraries;
- Its interpreter is easily embeddable in other applications;
- It is easy to wrap any API in python without loss of functionality.

4. Basic Architecture of the Query Engine

This section illustrates briefly the main components of the query engine, some of which are referred later in this document to help understand how CloudMdsQL queries work. The basic query engine architecture is depicted in Fig. 1. It consists of the following components:

- Query compiler/optimizer: performs query decomposition to a query execution plan. It may use a catalog for semantic analysis of the query and/or for estimating the costs of relational algebra operations, used by the optimizer.
- Query processor: executes the query using the generated execution plan. It interacts with wrappers in order to query the underlying data stores.
- Operator engine: used by the query processor to execute relational operators on the data obtained from the wrappers and produce the final result.
- Table Storage: stores intermediate and result data – in memory or on disk.
- Wrappers: each wrapper retrieves data from its data store and produces relations to be consumed by the leaf nodes of the execution plan.

The components exchange with each other the following data structures:

- CloudMdsQL query is the one that is passed to the query engine. It contains nested queries to local data stores that are passed by the core engine to the wrappers.
- Query execution plan is the result from the query decomposition, performed by the compiler/optimizer that is delivered to the query processor for execution. In its simplest form this is a tree structure, representing relational algebra operations, where the leaf nodes are references to tables, results from the execution of the named table expressions by the wrapper. Whenever nested table expressions are used in the query, arcs might exist between leaf nodes, making the execution plan a graph rather than a tree structure.
- Data store queries are passed by the query processor to the wrapper for execution against the data store. They conform to the data store's native query mechanism.

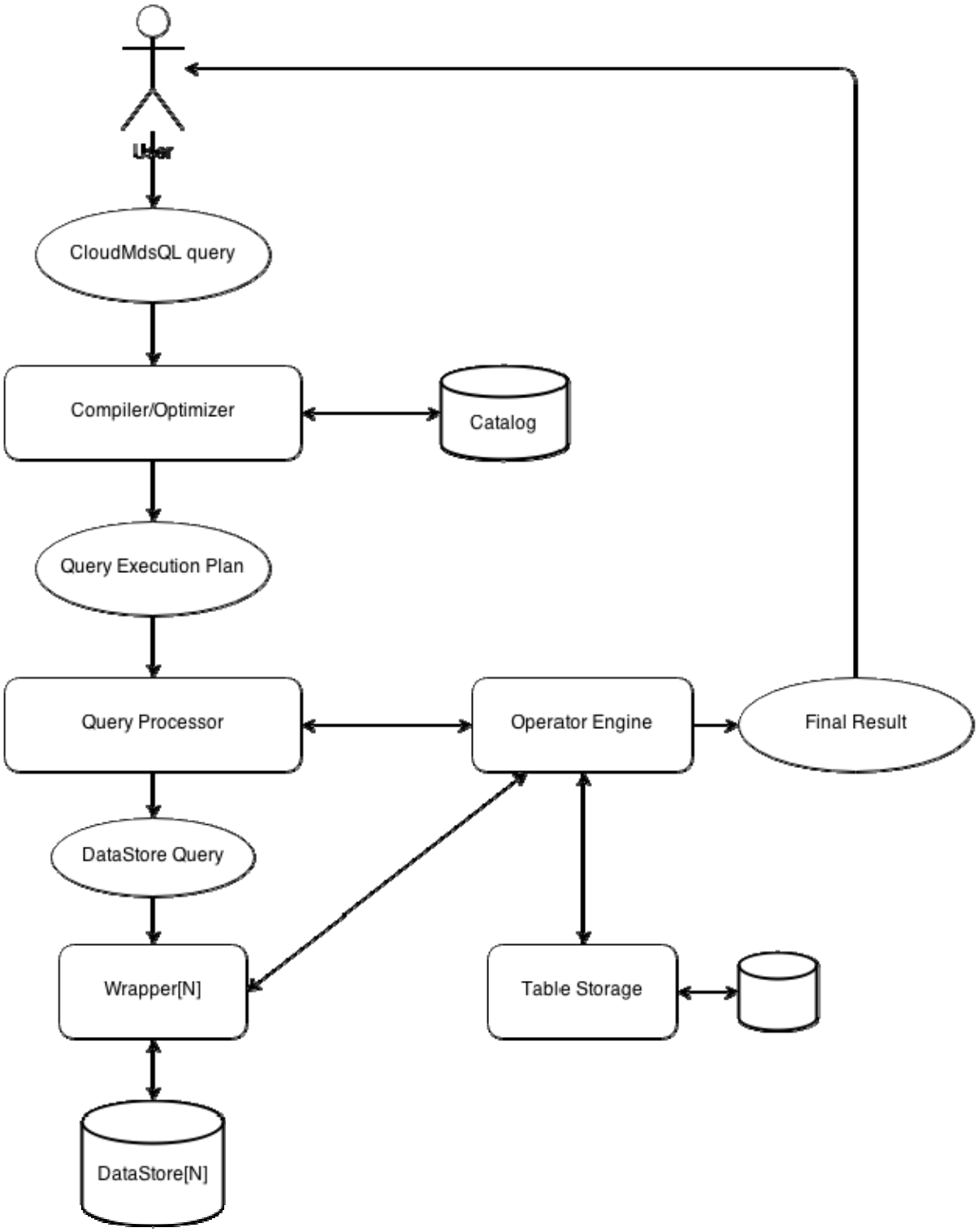


Fig.1. Basic architecture of the query engine

5. Query Language Concepts

The common query language concept introduces the notion of "table expressions", which are definitions of temporary (at query level) tables representing nested queries. A named table expression has a signature that defines the names and types of the attributes of the returned relation. Thus, each CloudMdsQL query is executed in the context of a kind of ad-hoc schema, formed by all named table expressions within the query. This approach fills the gap produced by the lack of a global schema and allows the query compiler to perform semantic analysis of the query.

5.1. Named Table Expressions

Named table expressions are defined in the header of a CloudMdsQL query, preceding the SELECT keyword, and are instantiated in the FROM clause and/or from the definitions of other named expressions. Several types of named table expressions are defined:

5.1.1. Native Named Table Expressions

Expressions of this type are referencing queries to data stores using their native query mechanism. They are executed in the context of a particular connection to a data store.

To distinguish native expressions, CloudMdsQL introduces ***native expression brackets***, which can be any of the following pair of opening / closing bracket symbols:
`{* ... *}, {< ... >}, {/ ... /}, {@ ... @}`

For simplicity, further in this document we will use only the first notation to represent native expression brackets, but all notations can be used interchangeably.

Syntax:
```
<table-name>(<attrname> <attrtype>, …)@<data-store> = {* <table-expr> *}
```

Where:

- `<table-name>` is the alias of the temporary table that will be referenced in the CloudMdsQL SELECT expression.
- `(<attrname> <attrtype>, …)` is the signature of the named table expression, which declares the names and types of the attributes of the result relation.
- `<data-store>` is a reference to the underlying data store which will be queried.
- `<table-expr>` is an expression, containing invocations to the native interface of the data store and produces a relation with the declared signature.
- The surrounding native expression brackets give information to the query engine not to process the contained `<table-expr>` but pass it as a black box to the corresponding wrapper.

5.1.2. SQL Named Table Expressions

Expressions of this type can instantiate named tables from the context of the current CloudMdsQL query but can also reference tables from a corresponding data store. These expressions are processed by the query engine. The `@<data-store>` declaration is optional and if not present, this means that the `<table-expr>` SQL expression references only named table instantiations from the context of the CloudMdsQL query. The expression code is surrounded by parentheses:

```
<table-name>(<attrname> <attrtype>, …)[@<data-store>] = ( <table-expr> )
```

Where:

- `<table-name>` is the alias of the temporary table that will be referenced in the CloudMdsQL SELECT expression.
- `(<attrname> <attrtype>, …)` is the signature of the named table expression, which declares the names and types of the attributes of the result relation.
- `<data-store>` is a reference to the underlying data store; optional.
- `<table-expr>` is an SQL SELECT statement that can combine data store tables with named tables.
- If `<data-store>` is specified, `<table-expr>` may be rewritten and split by the query optimizer into two queries – the first one will contain only references to data store tables and will be passed to the data store; the other one will reference only named table instantiations and will be executed in the context of the CloudMdsQL query; then both results will be integrated to form the result dataset.
- The surrounding parentheses (unlike the native expression brackets) give information to the compiler that the contained `<table-expr>` is an SQL expression that must be processed by the CloudMdsQL engine.

5.1.3. Python Named Table Expressions

Expressions of this type are not referencing a data source, but are executed in the context of the current CloudMdsQL query. The synopsis lacks the `@<data-store>` declaration and the expression code is surrounded by braces:

```
<table-name>(<attrname> <attrtype>, …) = {* <table-expr> *}
```

Where:

- `<table-name>` is the alias of the temporary table that will be referenced in the CloudMdsQL SELECT expression.
- `(<attrname> <attrtype>, …)` is the signature of the named table expression, which declares the names and types of the attributes of the result relation.
- `<table-expr>` is a block of Python code that is executed in the context of the CloudMdsQL query and may reference other named table expressions.
- In this type of named table expression, the contained `<table-expr>` is surrounded by native expression brackets, which means that the query engine will just pass it as a black box to the embedded Python interpreter, which is part of the operator engine.

5.2. Python Integration

To achieve interoperability between the embedded Python code and the query engine, special conventions are used. They are described briefly below and their usage is demonstrated for better clarity in the examples section.

5.2.1. Common Python Conventions

The keyword `yield` is used to append tuples to the result dataset and is used within the Python code of named table expressions the following way:

```
T1( x int, y string )@db1 = {*
  ...
  yield (1, 'abc')
  ...
*}
```

This line of code appends to the result dataset a tuple corresponding to the named table's signature. It uses the standard Python yield feature.

Another common convention is the usage of the special Python object `CloudMdsQL` that holds the context of the current CloudMdsQL query execution. It can be used in the Python code of a named table expression to instantiate other named expressions available in the context of the same query. Example:

```
T2( x int, len_y int )@db1 = {*
  ...
  for (x, y) in CloudMdsQL.T1:
    yield (x, len(y))
  ...
*}
```

5.2.2. DB Specific Python Objects

Other data store specific objects might also be referenced; e.g. when querying Dex, a `graph` object should be invoked from the Python code. These objects must be initialized and maintained by the corresponding wrapper. A CloudMdsQL user should be aware of the presence and the names of such objects in order to reference them from a CloudMdsQL query.

5.3. Named Action Expressions

Similarly to table expressions, the common query language makes use of "action expressions", which are composed of statements that perform data modification operations against the corresponding data store. They are used in data manipulation CloudMdsQL statements as opposed to named table expressions which are used in data query commands. CloudMdsQL data manipulation statements contain at least one EXECUTE clause, which may be preceded by named action expressions having the following syntax:

```
<action-name>[(<signature>)]@<data-store> = {* <action-expr> *} or
```

```
<action-name>[(<signature>)]@<data-store> = ( <action-expr> )
```

Where:

- `<action-name>` is the alias of the action expression.
- `<data-store>` is a reference to the underlying data store against which the action will be performed.
- `<action-expr>` can be one of the following:
  - SQL DML statement (INSERT, UPDATE or DELETE);
  - Expression, containing invocations to the native interface of the data store and performs data modification.
- If `<action-expr>` is surrounded by native expression brackets, it will be passed as a black box to the wrapper. If surrounded by parentheses, it will be processed by the query engine, which may rewrite the statement according to the usage of CloudMdsQL tables in it.
- If `<signature>` is specified, it can contain `REFERENCING` and/or `WITHPARAMS` clauses (see below).

5.4. Instantiating Other Named Table Expressions

As stated in the language requirements, CloudMdsQL must provide a mechanism for nesting queries – i.e. a named table/action expression must be able to instantiate other named table expressions available in the context of the same query. This is achievable in all types of expressions: in Python by invoking the `CloudMdsQL` object, and in SQL by simply referencing named tables directly in the FROM clause, often in combination with references to the data store's tables.

To build a relevant and adequate query execution plan, the query optimizer needs to identify all dependencies between table expressions. Therefore, for each named expression, the optimizer needs to know which other named tables it references. For SQL expressions, this dependency is discovered automatically because the query engine performs processing of the SQL expression. For native/Python expressions, however, since a black-box approach is used, the query engine doesn't perform any processing; therefore the referenced inside the expression named tables must be explicitly specified in the named expression's signature.

This must be done by adding the names of the referenced CloudMdsQL tables to the signature following the `REFERENCING` keyword:

```
T2( x int, len_y int
    REFERENCING T1 )@db1 =
{*
  ...
  for (x, y) in CloudMdsQL.T1:
    yield (x, len(y))
  ...
*}
```

If a table name is instantiated within a native or Python expression without being specified in the REFERENCING clause, the entity that processes the expression code (either a wrapper or the operator engine) must throw a run-time exception the first time it tries to reference the table.

With dependencies between named tables, the query execution plan is no longer a tree, but a graph structure. However, circular references are not allowed – if there is a cycle in the graph, the compiler will detect it and the query will fail to compile.

5.5. Parameterized Expressions

Named expressions can be parameterized, thus making table expressions behave like parameterized views and action expressions – like parameterized procedures. The names and types of the parameters must be declared in the signature following the `WITHPARAMS` keyword. Each parameter must be referenced inside the expression by a named placeholder. For SQL expressions, the placeholder is composed of the parameter name, prefixed by a dollar sign. For Python expressions, another special Python object will be used to refer to parameters. For native expressions, the used placeholder convention is implementation-specific and it is the wrapper's responsibility to identify parameter placeholders and replace them with actual values. However, for better readability, in this document we will use only the dollar-sign notation. Parameterized named table/action expressions need to be instantiated from other expressions or in a FROM clause by passing actual parameter values.

Example:

```
T1( x int, y string
    WITHPARAMS a string )@db1 =
(
  SELECT x, y FROM tbl WHERE id = $a
)
T2( x int, sqr_x int
    WITHPARAMS a int, b int )@db2 =
{*
  for x in range($a, $b):
    yield (x, x*x)
*}
...
SELECT T1.x, T1.y, T2.sqr_x
FROM T1('abc') JOIN T2(1,5) ON T1.x = T2.x
```

5.6. Storing Expressions

CloudMdsQL allows a (parameterized) table or action expression to be given a global name and stored in a global context in order to be referenced in several CloudMdsQL queries, similarly to SQL stored procedures/functions. Example:

```
CREATE NAMED EXPRESSION T1( ... )@db1 = {*
  ...
*}
```

6. Query Language Specification

This section will give an overview of the main CloudMdsQL commands while the language is formalized in detail in the Appendix.

6.1. Data Definition

Since there is no global schema in the common data model, respectively there are no data definition CloudMdsQL commands. The only command that can affect the global context is the CREATE NAMED EXPRESSION command, which can store the definition of a named table or action expression globally, so that it can be used in many CloudMdsQL queries. The syntax is:

```
CREATE NAMED EXPRESSION <named-expr>
```

where <named-expr> is either a named table expression or a named action expression as per the corresponding syntaxes described in the previous section.

To drop a stored named expression, the CloudMdsQL command is:

```
DROP NAMED EXPRESSION <expr-name>
```

Where <expr-name> is the name (string) of the stored expression to be dropped.

6.2. SELECT Query

The purpose of SELECT query statements in CloudMdsQL is to retrieve data from several data stores using embedded sub-queries (for each data store) and integrate the data to build the result dataset. The CloudMdsQL SELECT statement is a derivative of the standard SQL SELECT statement and supplements it with a header containing definitions of named table expressions:

```
[<named-table-expr> ...]
SELECT <target_list>
[<from_clause>]
[<where_clause>]
[<group_clause>]
[<having_clause>]
[<order_clause>]
[<limit_clause>]
```

Where:

- [<named-table-expr> ...] is an optional list of named table expressions as per the corresponding syntaxes described in the previous section. Names of table expressions must be unique within both the local (in the same query) and global (stored named expressions) context.
- <from_clause> is the regular SQL FROM clause containing references to named tables – global or ad-hoc, parameterized or not. If a table refers to a parameterized named table expression, the parameter values should be specified

in parentheses. The FROM clause can contain JOIN expressions, specifying explicit join order and conditions. The JOIN keyword can be preceded by the BIND keyword, which will explicitly instruct the query planner to use bind join.

- In the <where_clause> there can be specified a filter predicate expression. The query compiler will transform it to normal conjunctive form, thus determining the exact selection operations to be performed as part of the execution plan. The optimizer will then find the most appropriate place of each selection operation and push it down as much as possible in the execution plan tree. This optimization can finally result in rewriting sub-queries to data stores by adding WHERE clause conditions, if the optimizer finds an opportunity to increase the selectivity of the sub-query. However, only sub-queries defined with SQL named table expressions can benefit from such an optimization.
- The remaining clauses are exactly as they appear in SQL.

6.3. Data Manipulation

With data manipulation commands of CloudMdsQL the user can modify data in the data stores by using the native data manipulation mechanism of each data store. For SQL data stores this is done with embedded action expressions containing INSERT, UPDATE or DELETE command, while for NoSQL data stores the embedded action expressions perform invocations to the data stores' query APIs. An action expression can instantiate named table expressions which gives the flexibility for a single query to retrieve data from one (or more) data store, perform some transformations on the data and then use it to update another data store, similarly to a typical ETL task. Moreover, a single CloudMdsQL command can perform data manipulation against several data stores.

The basic construct of a data manipulation command in CloudMdsQL is the EXECUTE clause, which can be more than one within a single command. An execute clause specifies a data store and an action expression that will be executed against the data store. Action expressions can be either inline or named, which is described below.

6.3.1. Executing Inline Action Expressions

Syntax:

```
EXECUTE@<data-store> {* <action-expr> *} or
EXECUTE@<data-store> ( <action-expr> )
```

With this syntax of the EXECUTE clause the action expression specified inside braces (native) or parentheses (SQL) will be executed against the specified data store.

6.3.2. Executing Named Action Expressions

Syntax:

```
EXECUTE <action-expr-name> [ ( <param-values> ) ]
```

With this syntax of the EXECUTE clause the action expression named <expr-name> will be executed. This must be a reference to either a named action expression whose definition is given within the same query preceding the EXECUTE keyword, or a stored named action expression. Parameter values can also be given.

6.4. Transaction Management

An application can execute several CloudMdsQL statements in a transaction block which appears in isolation across all data stores thanks to the holistic transaction management subsystem. Normally the application controls a transaction, i.e. it holds the "transaction context", which consists of transaction ID and start timestamp. In this case the application invokes the transaction management API methods.

However the user is given the possibility to invoke the transaction management interface directly via CloudMdsQL, which supports the traditional transaction management commands `START TRANSACTION`, `COMMIT` and `ROLLBACK`. In this case the common query engine is the one that holds the transaction context internally and controls the transaction transparently from the user, associating it only to the current session.

7. Interfacing the Data Stores

As stated in the previous sections, whenever a CloudMdsQL query is executed, the query engine prepares a set of native queries that need to be executed against the data stores. The engine then passes each query to the corresponding wrapper, which is responsible for the following:

- To represent transparently its underlying data store;
- The execution of native sub-queries against the data store, for which there are two possibilities:
  - Server-side execution: The wrapper passes the query to the data store for remote execution (e.g. SQL);
  - Client-side execution: The wrapper executes the query locally, accessing the data store through a client library and API (e.g. DEX and its Python API);
- To guarantee that the retrieved data matches the number and types of columns, specified in the signature of the expected dataset in the CloudMdsQL query;
- To deliver the retrieved datasets to the operator engine;
- To be able to instantiate other named table expressions, hence to access intermediate relations from the table storage component during execution.

7.1. SQL Compatible Data Stores

In order to be queried through CloudMdsQL each data store needs to expose a query interface capable of producing relational datasets. CloudMdsQL conforms to the SQL standard and its compiler generates an execution tree of relational algebra operators. Whenever an SQL table expression is used as a nested query against a data store, it is considered as a sub-select statement and hence is transformed into a sub-tree in the query execution plan. Thus, each SQL table expression can be subject to further transformations and may be possibly rewritten by the optimizer before submitted for execution to the data store. Therefore, it is recommended that a data store exposes an SQL like interface (whenever possible without compromising the functionality), because thus the CloudMdsQL engine will be able to perform optimizations of the query execution plan, e.g. pushing selections, projections and join operations as down the tree as possible, performing bind-joins, etc. It might also be possible to access one data store through two wrappers via both SQL and native query interface.

7.2. Requirements for Native Queries

In a CloudMdsQL query, to write native named table expression sub-queries against SQL incompatible data stores, embedded blocks of native query invocations are used. Although in this document we refer to such queries as Python based, it is possible to embed queries in any other language, as long as they fulfil the following requirements:

- Each query must produce a relation according to the common data model; the corresponding wrapper is then responsible to convert the data set to match the declared signature, if needed;
- In order to fulfil the requirement for nested tables support, the language should provide mechanism to instantiate other named tables; Python analogue: the

`CloudMdsQL` object. This is the major drawback of embedding server-side native query language, other than SQL. However, if nesting is not needed, such native language can still be used, but it is recommended that the data store provide an alternative query interface that supports nesting, e.g. SQL.

For example, it is possible for a wrapper to represent an MDX data store, but the application programmer must be aware of the following issues:

- An MDX query with expanded members of an axis on columns has variable number of columns of the result dataset, which makes it impossible to match a table expression signature with fixed columns. However, if the last column of the named table expression is of type `dictionary`, the wrapper can encapsulate all variable-length data into `dictionary` objects and thus to comply with the declared signature.
- There will be no way to reference other named tables inside an MDX native query, so nesting another sub-query into an MDX sub-query will not be supported.

8. Examples

This section will cover the language concepts through several examples.

8.1. Example 1. Understanding Python usage to produce relations

The following query contains three Python named table expressions that simply generate tuples of hardcoded data. The SELECT statement then performs relational algebra operations on the synthesized datasets to find 'All publications of scientists from INRIA reviewed in 2013 and their reviewers'.

```
scientists( name string, affiliation string, country string ) = {*
  yield ('Ricardo', 'UPM', 'Spain')
  yield ('Martin', 'CWI', 'Netherlands')
  yield ('Patrick', 'INRIA', 'France')
  yield ('Boyan', 'INRIA', 'France')
  yield ('Larri', 'UPC', 'Spain')
  yield ('Rui', 'INESC', 'Portugal')
*}

pubs( id int, title string, author string ) = {*
  yield (1, 'Snapshot isolation in ...', 'Ricardo')
  yield (5, 'Principles of DDBS', 'Patrick')
  yield (9, 'Graph DBs', 'Larri')
*}

reviews( pub_id int, date timestamp, reviewer string ) = {*
  yield (1, '2012-11-18', 'Martin')
  yield (5, '2013-02-28', 'Rui')
  yield (5, '2013-02-24', 'Ricardo')
  yield (9, '2013-01-19', 'Patrick')
*}

SELECT pubs.id, pubs.title, pubs.author, reviews.reviewer
FROM pubs
  JOIN reviews ON pubs.id = reviews.pub_id
  JOIN scientists ON pubs.author = scientists.name
WHERE scientists.affiliation = 'INRIA'
  AND Year(reviews.date) = 2013;
```

The result is:

id	title	Author	reviewer
5	Principles of DDBSs	Patrick	Rui
5	Principles of DDBSs	Patrick	Ricardo

8.2. Example Databases

In the above example we used hardcoded tables just to demonstrate the Python usage. For the rest of the examples we will retrieve data from three different databases and then integrate the results. We assume the following databases:

DB1 is a relational (e.g. MonetDB) database storing information about scientists and their publications in the following tables:

Scientists:

Name	Affiliation	Country
Ricardo	UPM	Spain
Martin	CWI	Netherlands
Patrick	INRIA	France
Boyan	INRIA	France
Larri	UPC	Spain
Rui	INESC	Portugal

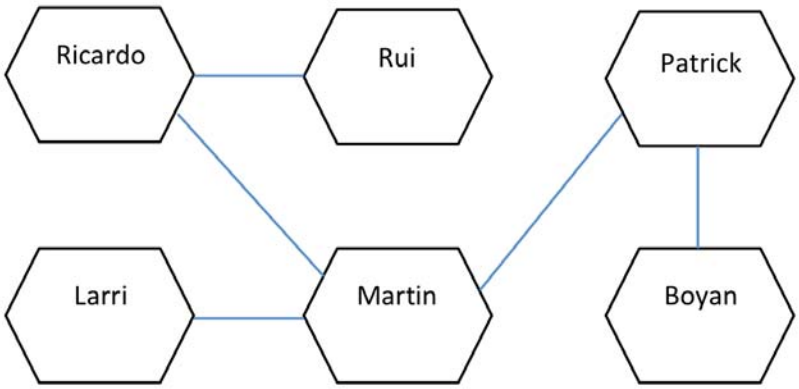
Pubs:

ID	Title	Author	Date
1	Snapshot Isolation	Ricardo	2012-11-10
5	Principles of DDBSs	Patrick	2011-02-18
9	Graph DBs	Larri	2013-01-06

DB2 is a document (e.g. MongoDB) database containing the following collection of paper reviews:

```
Reviews (
  {pub_id: 1, reviewer: 'Martin', date: '2012-11-18', review: '… text …'},
  {pub_id: 5, reviewer: 'Rui', date: '2013-02-28', review: '… text …'},
  {pub_id: 5, reviewer: 'Ricardo', date: '2013-02-24', review: '… text …'},
  {pub_id: 9, reviewer: 'Patrick', date: '2013-01-19', review: '… text …'}
)
```

DB3 is a graph database representing a social network with nodes representing persons and 'friend-of' links between them:



8.3. Example 2. Integrate results from a SQL and a document DB

We assume that the wrapper of DB2 uses the PyMongo library and holds a Python object named `db` that references the MongoDB database (a Python dictionary to all document collections). The following CloudMdsQL query again finds 'All publications of scientists from INRIA reviewed in 2013 and their reviewers':

```
pubs_I( id int, title string, author string )@DB1 = {*
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = 'INRIA'
*}

reviews_2013( pub_id int, reviewer string )@DB2 = {*
  filter = {'date': {'$and': {'$gte': '2013-01-01'},{'$lt': '2014-01-01'}}}
  for review in db['Reviews'].find( filter ):
    yield (review['pub_id'], review['reviewer'])
*}

SELECT pubs_I.id, pubs_I.title, pubs_I.author, reviews_2013.reviewer
FROM pubs_I
  JOIN reviews_2013 ON pubs_I.id = reviews_2013.pub_id;
```

The result is:

id	title	Author	reviewer
5	Principles of DDBSs	Patrick	Rui
5	Principles of DDBSs	Patrick	Ricardo

8.4. Example 3. Nested SQL queries and bind-join

This example achieves the same result as the previous one, but makes use of bind-join for better efficiency. Although in this example the bind-join is explicitly typed, in practice the CloudMdsQL query engine is supposed to automatically discover opportunities for performing bind-join. Note that `pubs_I` is now a SQL named table expression (surrounded by parentheses), not native as in the previous example. This is because the expression needs processing by the CloudMdsQL engine; it can't be passed as is to the data store because it contains references to the CloudMdsQL named table `reviews_2013`.

```
reviews_2013( pub_id int, reviewer string )@DB2 = {*
  filter = {'date': {'$and': {'$gte': '2013-01-01'},{'$lt': '2014-01-01'}}}
  for review in db['Reviews'].find( filter ):
    yield (review['pub_id'], review['reviewer'])
*}

pubs_I( id int, title string, author string )@DB1 = (
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = 'INRIA'
  AND pubs.id IN (SELECT pub_id FROM reviews_2013)
)
```

```
SELECT pubs_I.id, pubs_I.title, pubs_I.author, reviews_2013.reviewer
FROM pubs_I
  JOIN reviews_2013 ON pubs_I.id = reviews_2013.pub_id;
```

The result is:

id	title	Author	reviewer
5	Principles of DDBSs	Patrick	Rui
5	Principles of DDBSs	Patrick	Ricardo

8.5. Example 4. Native queries – Invoke Dex API

We add another named table `friends_P`, which has the goal to 'Find all friends and friends-of-friends of Patrick' by querying the graph database through its Python interface. All we need for the purpose is to have a Python interface to Dex objects and methods. The dataset produced by the named table `friends_P` is a relation containing all friends of Patrick and their level of friendship (friend or friend-of-friend). Then, in combination with the named tables from the previous example, we finally find 'All publications of scientists from INRIA, reviewed in 2013 by friends of Patrick':

```
pubs_I( id int, title string, author string )@DB1 = (
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = 'INRIA'
)

reviews_2013( pub_id int, reviewer string )@DB2 = {*
  filter = {'date': {'$and': {'$gte': '2013-01-01'},{'$lt': '2014-01-01'}}}
  for review in db['Reviews'].find( filter ):
    yield (review['pub_id'], review['reviewer'])
*}

friends_P( friend string, level string )@DB3 = {*

  nameType = graph.FindType('NAME');
  friendType = graph.FindType('FRIEND');

  nodePatrick = graph.Select( nameType, 'Patrick' )
  friends = graph.Neighbours( nodePatrick, friendType, Any )
  it = friends.Iterator()
  while it.HasNext():
    friendName = graph.GetAttribute( it.Next(), nameType )
    yield (friendName, 'Friend')

  friends = graph.Neighbours( friends, friendType, Any )
  it = friends.Iterator()
  while it.HasNext():
    friendName = graph.GetAttribute( it.Next(), nameType )
    yield (friendName, 'FriendOfFriend')

*}

SELECT pubs_I.id, pubs_I.title, pubs_I.author, reviews_2013.reviewer,
    friends_P.level
FROM pubs_I
  JOIN reviews_2013 ON pubs_I.id = reviews_2013.pub_id
  JOIN friends_P ON reviews_2013.reviewer = friends_P.friend;
```

The result is:

id	title	author	reviewer	level
5	Principles of DDBSs	Patrick	Ricardo	FriendOfFriend

8.6. Example 5. Nested queries – instantiating named table expression inside Python

In this example it will be demonstrated the usability of nested queries in native named table expressions. We define the SQL named table `pubs_I_revs_2013`, which retrieves 'All publications of scientists from INRIA reviewed in 2013 and their reviewers'. Then, we will discover conflicts of interests among these publications by querying the graph database for friendships between author and reviewer, which will be the result of the named table `conflicts_I`.

The thing to notice here is the usage of the `CloudMdsQL` object, which makes our query engine execute a nested query in the same context, instantiating the named table expression `pubs_I_revs_2013`, the results from which are then iterated in the Python code to check for possible friendships in the graph database.

To show a different way to query MongoDB, in this example we will assume that DB2_S is an alias to an SQL compatible wrapper for the same document database DB2. In this case, the table `reviews_2013` will be expressed using the SQL interface to MongoDB.

```
pubs_I( id int, title string, author string )@DB1 = (
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = 'INRIA'
)

reviews_2013( pub_id int, reviewer string )@DB2_S = (
  SELECT pub_id, reviewer
  FROM reviews
  WHERE year(date) = 2013
)

pubs_I_revs_2013( id int, author string, reviewer string ) = (
  SELECT pubs_I.id, pubs_I.author, reviews_2013.reviewer
  FROM pubs_I
    JOIN reviews_2013 ON pubs_I.id = reviews_2013.pub_id
)

conflicts( pub_id int, reviewer string, level string
    REFERENCING pubs_I_revs_2013)@DB3 =
{*
  nameType = graph.FindType('NAME');
  friendType = graph.FindType('FRIEND');

  for (id, A, R) in CloudMdsQL.pubs_I_revs_2013:
    nodeA = graph.Select( nameType, A )
    nodeR = graph.Select( nameType, R )
    sp = SinglePairShortestPath( graph, nodeA, nodeR )
    sp.AddEdgeType( friendType, Any )
    sp.SetMaximumHops( 2 )
    sp.Run()
```

```
    if sp.Exists():
      if sp.GetCost() == 1:
        conflict = 'Friend'
      else:
        conflict = 'FriendOfFriend'
      yield (id, R, conflict)

*}

SELECT pubs_I.id, pubs_I.title, pubs_I.author,
       conflicts.reviewer, conflicts.level
FROM pubs_I
  JOIN conflicts ON pubs_I.id = conflicts.pub_id;
```

The result is:

id	title	author	reviewer	level
5	Principles of DDBSs	Patrick	Ricardo	FriendOfFriend

8.7. Example 6. Parameterized expressions

Here we achieve the same results as in the previous example but make use of parameters. The Python code for the table conflicts is reduced to show only the difference with the one in the previous example.

```
pubs( id int, title string, author string
    WITHPARAMS aff string )@DB1 =
(
  SELECT pubs.id, pubs.title, pubs.author
  FROM pubs
    JOIN scientists ON pubs.author = scientists.name
  WHERE scientists.affiliation = $aff
)

reviews( pub_id int, reviewer string
    WITHPARAMS year int )@DB2 =
(
  SELECT pub_id, reviewer FROM reviews WHERE year(date) = $year
)

pubs_revs( id int, author string, reviewer string
    WITHPARAMS aff string, year int ) =
(
  SELECT p.id, p.author, r.reviewer
  FROM pubs($aff) p
    JOIN reviews($year) r ON p.id = r.pub_id
)

conflicts( pub_id int, reviewer string, level string
    REFERENCING pubs_revs
    WITHPARAMS aff string, year int )@DB3 =
{*
  ...
  for (id, A, R) in CloudMdsQL.pubs_revs($aff, $year):
    ...
*}

SELECT p.id, p.title, p.author, c.reviewer, c.level
FROM pubs('INRIA') p
  JOIN conflicts('INRIA', 2013) c ON p.id = c.pub_id;
```

8.8. Example 7. Data manipulation

To demonstrate how data can be modified through CloudMdsQL, we will store the retrieved data from the previous example in the table 'conflicts' in the relational database DB1, using an inline action expression. Note that in the INSERT statement, in order to avoid ambiguity, we put the prefix `CloudMdsQL` before the named table `conflicts`, while the target table has the same name but without the prefix, which means that it is located in the data store. For simplicity, only the signatures of the named table expressions will be shown, since the very expressions are the same as in the previous example:

```
pubs( id int, title string, author string WITHPARAMS aff string )@DB1 = (
  ...
)

reviews( pub_id int, reviewer string WITHPARAMS year int )@DB2 = (
  ...
)

pubs_revs( id int, author string, reviewer string WITHPARAMS aff string,
year int ) = (
  ...
)

conflicts( pub_id int, reviewer string, level string
    REFERENCING pubs_revs
    WITHPARAMS aff string, year int )@DB3 =
{*
  ...
*}

EXECUTE@DB1 (
  INSERT INTO conflicts
  SELECT p.id, p.title, p.author, c.reviewer, c.level
  FROM pubs('INRIA') p
    JOIN CloudMdsQL.conflicts('INRIA', 2013) c ON p.id = c.pub_id
);
```

Alternatively, we can also parameterize the above action expression and execute it as a named expression the following way:

```
...
...
...
...

store_conflicts( WITHPARAMS aff string, year int )@DB1 = (
  INSERT INTO conflicts
  SELECT p.id, p.title, p.author, c.reviewer, c.level
  FROM pubs($aff) p
    JOIN CloudMdsQL.conflicts($aff, year) c ON p.id = c.pub_id
)

EXECUTE store_conflicts('INRIA', 2013);
```

9. Appendix A. Data Types

The following table displays the currently supported data types in CloudMdsQL with descriptions, type constructors and examples:

Type name and synonyms	Description	Type constructor	Examples
integer int	Integer number value	Decimal integer or 0x followed by hexadecimal integer	10 -9876 0xB84C
float double	Double precision floating point number value	Decimal floating point number with dot (.) as the decimal symbol	0.678 -9.521
string varchar	Variable length character string	Character string surrounded by quotes. Quotes, CR and LF are escaped with backslashes	'string' 'hello\nworld' 'Say \'hello\'''
timestamp	Date and time	Timestamp of the format: YYYY-MM-DD hh:mm:ss.xxx surrounded by quotes and prefixed by T	T'2014-03-31 12:20:00'
binary	Variable length byte array	Hexadecimal byte array surrounded by quotes and prefixed by H or Bit array surrounded by quotes and prefixed by B	H'A012C98BDE' B'10011010011'
array	Variable length array of values of any of the herewith supported types	Comma separated list of values surrounded by brackets	[1, 3, {'a': 5, 'b':7}]
dictionary json	Set of key-value pairs; key is string with unique values within the dictionary and value is of any of the herewith supported types	Comma separated list of key:value pairs surrounded by curly braces	{'id':1, 'version':'1.1', 'authors': ['Patrick', 'Boyan']}

Table 1. CloudMdsQL Data Types

10. Appendix B. CloudMdsQL Grammar

The CloudMdsQL context-free grammar is formalized in this section. This grammar formalization uses the following notations:

Tokens:

```
UPPERCASE     keyword
lower_case    nonterminal symbol, left-hand side of exactly one rule
Camel_Case    terminal symbol, described below
'x'           single character symbol
stmt          is the starting nonterminal symbol
```

Special grammar symbols:

```
:=   production rule symbol
|    disjunction
()   subexpression
*    zero or more successive occurrences of the preceding construct
+    one or more successive occurrences of the preceding construct
[]   optional construct
;    end of production rule
```

Terminals:

```
Identifier    matches the regular expression [A-Za-z_][A-Za-z_0-9]*
Type          name or synonym of any CloudMdsQL type
Ext_Code      any text surrounded by native expression brackets
Operation     any of the listed below binary operations
Placeholder   dollar sign followed by Identifier
Value         type constructor of any CloudMdsQL type
Natural       matches the regular expression [1-9][0-9]*
```

Binary operations (in order of precedence):

```
*, /, %
+, -, ||
=, <, >, <=, >=, <>
AND
OR, XOR
```

CloudMdsQL Grammar:

```
stmt                 := dql_stmt | dml_stmt | ddl_stmt | trans_stmt;

dql_stmt             := named_table_expr* select_stmt;
dml_stmt             := (named_table_expr | named_action_expr)* execute_stmt+;
ddl_stmt             := create_expr_stmt | drop_expr_stmt;
trans_stmt           := START TRANSACTION | COMMIT | ROLLBACK;

named_table_expr     := table_signature ['@' datastore] table_expression;
table_signature      := table_name '(' arg_list [referencing_clause] [withparams_clause] ')';
table_expression     := Ext_Code | '(' select_stmt ')';
```

```
select_stmt          := SELECT target_list [from_clause] [where_clause] [group_clause]
                               [having_clause] [order_clause] [limit_clause];
target_list          := target_col (',' target_col)*;
target_col           := expression [AS col_alias];

from_clause          := FROM from_item (',' from_item)*;
from_item            := (table_ref | sub_select) [[AS] table_alias]
                      | join_expr
                      | '(' from_item ')';
join_expr            := from_item [BIND] JOIN from_item ON expression;

where_clause         := WHERE expression;
group_clause         := GROUP BY expression (',' expression)*;
having_clause        := HAVING expression;
order_clause         := ORDER BY expression [ASC|DESC] (',' expression [ASC|DESC])*;
limit_clause         := LIMIT Natural;

named_action_expr    := action_signature '@' datastore action_expression;
action_signature     := action_name [ '(' [referencing_clause] [withparams_clause] ')' ];
action_expression    := Ext_Code | '(' action_stmt ')';

arg_list             := attr_name Type (',' attr_name Type)*;
name_list            := Identifier (',' Identifier)*;

referencing_clause   := REFERENCING name_list;
withparams_clause    := WITHPARAMS arg_list;

execute_stmt         := EXECUTE '@' datastore action_expression
                      | EXECUTE action_ref
                     ;
insert_stmt          := INSERT INTO table_ref [ '(' column_ref [',' column_ref] ')' ]
                        ( select_stmt | VALUES value_list );
update_stmt          := UPDATE table_ref SET update_col (',' update_col)* [where_clause];
update_col           := column_ref '=' expression;
delete_stmt          := DELETE FROM table_ref [where_clause];

create_expr_stmt     := CREATE NAMED EXPRESSION named_table_expr | named_action_expr;
drop_expr_stmt       := DROP NAMED EXPRESSION Identifier;

expression           := column_ref | value | sub_select | function_call | case_expr
                      | Placeholder
                      | '(' expression ')'
                      | expression Operation expression
                      | (NOT | '-' | '+') expression
                      | expression IS [NOT] NULL
                      | expression IN ( sub_select | value_list )
                      ;

case_expr            := CASE [expression] when_clause+ [else_clause] END;
when_clause          := WHEN expression THEN expression;
else_clause          := ELSE expression;

sub_select           := '(' select_stmt ')'
value_list           := '(' value (',' value)* ')';
action_stmt          := insert_stmt | update_stmt | delete_stmt;

attr_name            := Identifier;
table_name           := Identifier;
datastore            := Identifier;
col_alias            := Identifier;
table_alias          := Identifier;

schema_ref           := Identifier;
table_ref            := ( [schema_ref '.'] Identifier )
                      | ( table_name ['(' expression (',' expression)* ')'] );
action_ref           := action_name ['(' expression (',' expression)* ')'];
function_ref         := [schema_ref '.'] Identifier;
column_ref           := [table_ref '.'] Identifier ('.' Identifier)*;

function_call        := function_ref '(' [expression (',' expression)*] ')';
value                := NULL | TRUE | FALSE | Value;
```

11. References

[1]. C. Binnig, R. Rehrmann, F. Faerber, R. Riewe. FunSQL: it is time to make SQL functional. EDBT/ICDT Workshops, 2012.
[2]. S. Danforth, P. Valduriez. A FAD for Data-Intensive Applications. IEEE Trans. on Data and Knowledge Engineering, 4(1):34-51, 1992.
[3]. A. Doan, A., Halevy, Z. Ives. Principles of Data Integration. Morgan Kaufmann, 2012.
[4]. P. Haase, T. Mathäß, M. Ziller. An evaluation of approaches to federated query processing over linked data. International Conference on Semantic Systems (I-SEMANTICS), 2010.
[5]. E. Meijer, B. Beckman, G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. ACM SIGMOD Conference 2006.
[6]. T. Özsu, P. Valduriez. Principles of Distributed Database Systems – Third Edition. Springer, 850 pages, 2011.
[7]. P. Valduriez, S. Danforth. Functional SQL, an SQL Upward Compatible Database Programming Language. Information Sciences, 62(3):183-203, 1992.
[8]. C. M. Wyss, E.L. Robertson. Relational Languages for Metadata Integration. ACM Trans. On Database Systems, 30(2):624-660, 2005.
[9]. Laura M. Haas, Donald Kossmann, Edward L. Wimmers, Jun Yang. Optimizing Queries across Diverse Data Sources. VLDB 1997: 276-285