CoherentPaaS

Coherent and Rich PaaS with a Common Programming Model

ICT FP7-611068

Architecture of the Query Engine for the Common Query Language

D3.2

October, 2014

Document Information

Scheduled delivery	30.09.2014
Actual delivery	24.10.2014
Version	1.0
Responsible Partner	INRIA

Dissemination Level:

PU Public
PP Restricted to other programme participants (including the Commission)
RE Restricted to a group specified by the consortium (including the Commission)
CO Confidential, only for members of the consortium (including the Commission)

Revision History

Date	Editor	Status	Version	Changes
22.09.2014	N. Martínez	Draft	0.1	Skeleton
07.10.2014	D. Dominguez	Draft	0.2	Query engine contents
07.10.2014	B. Kolev	Draft	0.2	Query compiler contents
07.10.2014	J. Orlando	Draft	0.2	Wrapper contents
13.10.2014	D. Dominguez	Draft	0.3	First full draft
14.10.2014	J. Orlando D. Dominguez B. Kolev	Draft	0.4	Review and minor editions
15.10.2014	D Dominguez	Draft	0.5	Draft to reviewers
16.10.2014	M. Kersten L. Cortesão		0.6	Review
21.10.2014	D. Dominguez J. Orlando B. Kolev	Draft	0.7	Updates from first review
22.10.2014	J. Orlando B. Kolev	Draft	0.8	Minor editions
24.10.2014	R. Jimenez	Final	1.0	Final coordinator revision

Contributors

Boyan Kolev (INRIA), Norbert Martínez (Sparsity), David Domínguez (Sparsity), Ricard Gavaldà (UPC), Marta Perez (UPC), Josep Lluis Larriba, (Sparsity), Jose Orlando Pereira (INESC)

Internal Reviewers

Martin Kersten (MonetDB Solutions), Luis Cortesão (PTIN)

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
CloudMds	Cloud Multi-data store
CQA	CloudMds Query Algebra
CQE	CloudMds Query Engine
JDBC	Java Database Connectivity

Table of Contents

1. Executive Summary

Providing massive data processing capabilities in the cloud is a major trend in the design of data management solutions deployed on the cloud. The experience of the latest years is that no single data management system is the silver bullet for data processing, where all the data needs can be mapped. Currently, companies are using a variety of data solutions ranging from relational databases to NoSQL data stores, which come in multiple flavors such as graph databases, key-value data stores, array data stores, analytical cloud frameworks, document databases, data stream systems, etc. CoherentPaaS provides a software infrastructure to allow efficient and easy to program communication among this multitude of data management systems. In this deliverable, we describe the architecture of the query engine that interconnects the data stores.

The query engine is central piece that coordinates the execution of queries in CoherentPaaS. This module executes queries in the CloudMdsQL language (see D3.1), which defines a syntax to mix the operations among the data repositories. The data in the query engine is modeled as tabular data: sets of tuples with a fixed number of attributes. This model is simple enough to allow importation and exportation of data from NoSQL data representations.

Clients connecting to the CoherentPaaS infrastructure will have the impression that all systems act as a single database. In order to provide such a feeling from the client perspective, we will provide a JDBC connector for the CoherentPaaS infrastructure of CoherentPaaS. JDBC is one of the standard methods to connect to a database system, and thus clients will connect to the database like a regular database system. However, user queries will be able to take full advantage of the different repositories in the CoherentPaaS infrastructure by introducing SQL and NoSQL statements.

This deliverable contains the description of the architecture of the query engine for the common query language and its interaction with the data stores. In the next year of the project, we will implement this architecture and build the prototype for the query engine and compiler. This deliverable is structured as follows:

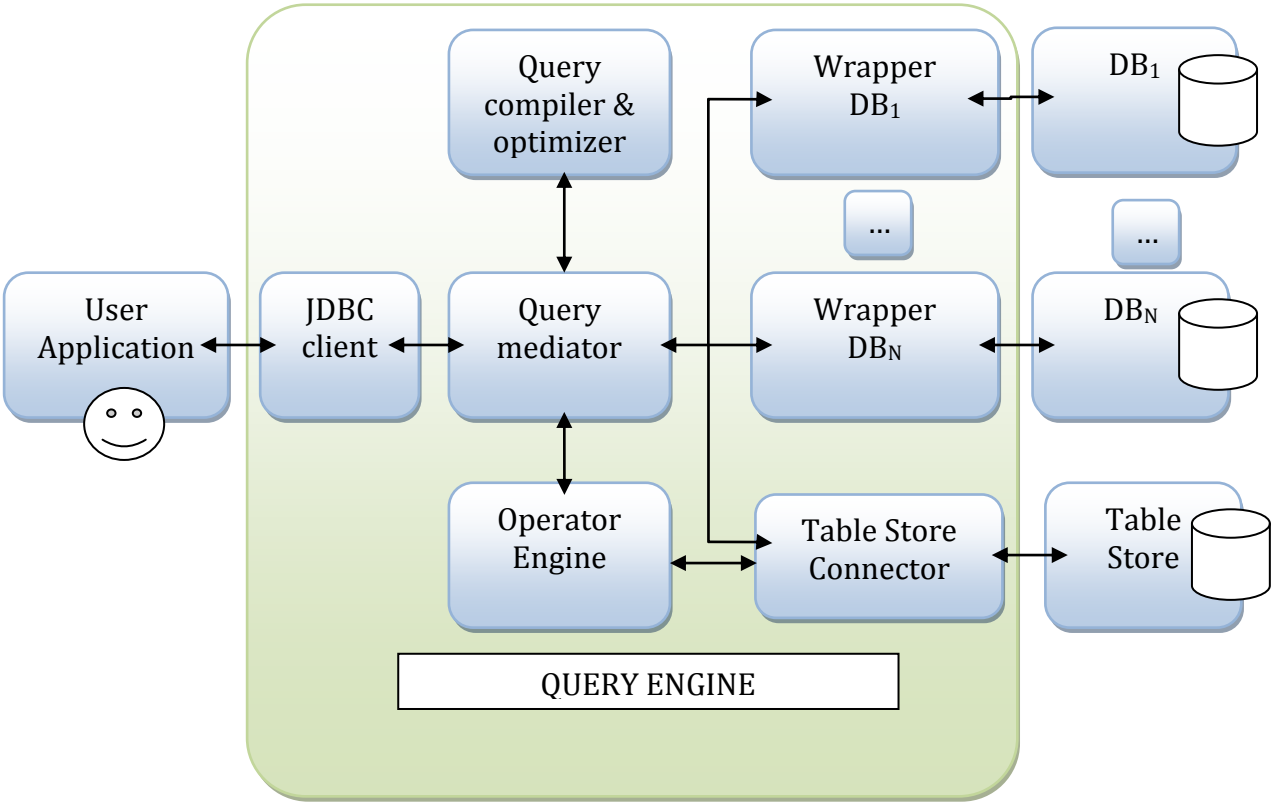
- In Section 2, we describe the global query engine architecture. The global Query Mediator performs all the coordination activities among the modules and is the central block of the CoherentPaaS architecture. The Query Engine is able to compute queries by forwarding them to the Data Stores, and also by combining results using the Operator Engine. The Operator Engine is able to execute programs that combine CloudMds Query Algebra (CQA) operators in a query plan. The Query Engine has also a module, called Table Store, which saves the tabular results of queries into tables. The tables contain either the query results or temporary results that will be further processed by the Operator Engine or any of the data stores in the CoherentPaaS ecosystem. We select one of the database providers of the consortium, MonetDB, as the storage of the Table Store.
- In Section 3, we explain the query compiler architecture. This module performs the compilation of the queries. The compilation process starts with the syntactic and semantic validation of a given query, and then performs a sequential set of optimization procedures that rewrite the user query to equivalent statements that are faster to compute. The query plan is finally converted to CQA operations

that will control the execution flow of the data between the Data Stores and the Operator Engine.

- In Section 4, we describe the architecture of the Operator Engine and the set of operations that will be implemented in the project. The Operator Engine implements a set of operations that build CQA programs. These operations process and combine the results provided by the different data stores.
- In Section 5, we describe the communication interface of the query engine with the data store by means of a wrapper infrastructure. This interface is implemented using a single API, which unify the interaction between the Query Mediator and the data stores. Using this approach, we ensure that data providers not part of the CoherentPaaS project can attach their database to the CoherentPaaS platform by only implementing the wrapper interface. In this section, we discuss the calls to the interface and the logic that the data store provider must implement.

2. CloudMds Query Engine Architecture

The Cloud multi-data store Query Engine of CoherentPaaS will be integrated with a very scalable transactional processing system that provides full ACID transactions over arbitrary sets of cloud data stores. CoherentPaaS will use a mediator/wrapper architecture. The Query Mediator is the component that performs communication between the clients that implement the user functionalities, and the set of database engines in CoherentPaaS. We depict a schema of the modules:



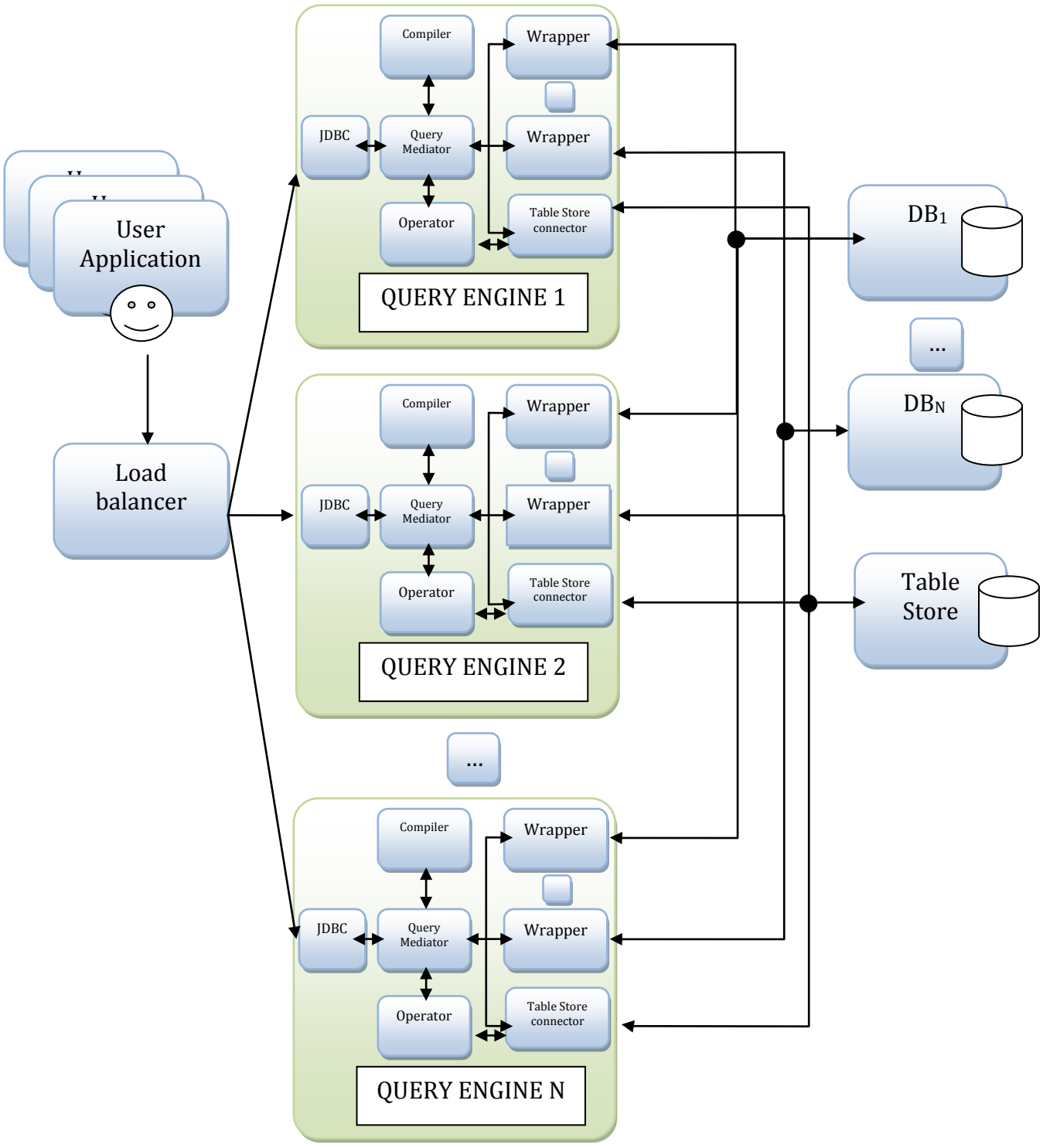
The main modules of the query engine are the following:

- JDBC client: Interfaces to provide access to the database architecture.
- Query compiler & optimizer: Module that transforms a CloudMdsQL query to the CQA algebra language. The optimizer will perform a set of processes that rewrite the original query to an equivalent query in CloudMdsQL language with a smaller expected execution time.
- Query Mediator: Controller class that coordinates the client communication, query compilation, execution of query operators and access to databases of CoherentPaaS. The query mediator will contain an instance of the local transaction manager that will act as a proxy for the holistic transaction manager.
- Table Store: Storage of temporary results of the queries.
- Operator Engine: component that implements relational operators in order to combine and process the results of the data stores and the Table Store.

- DB_i: database provider available in the CoherentPaaS platform. This includes both relational and NoSQL data repositories.
- Wrappers: interfaces that perform the connection between the Query Mediator and the database storages.

The first implementation of the query engine will be single instance. However, for scalability purposes, the query engine will be extended to a scalable system in the cloud. The architecture will replicate the query engine in multiple computing instances, in which each instance has all the capabilities to run a query. These instances will not share resources, but will access to any of the data stores available in the cloud. The access to these data stores will keep a strong coherence with the synchronization provided by the holistic transaction manager.

A load balancer will redirect the incoming new client queries to the query engine instance that is less loaded at that moment. Once a query has been compiled in an instance of the query engine, the user application will send the query to that query engine. After the evaluation of the first prototype, we will evaluate if it is necessary to apply a dynamic load balancing policy that allows queries to relocate among the query engines once they are compiled. The instances of the query engine will execute incoming queries as described for the case of the single query engine. The query engine instances will connect to the shared pool of databases through the corresponding data store connectors.

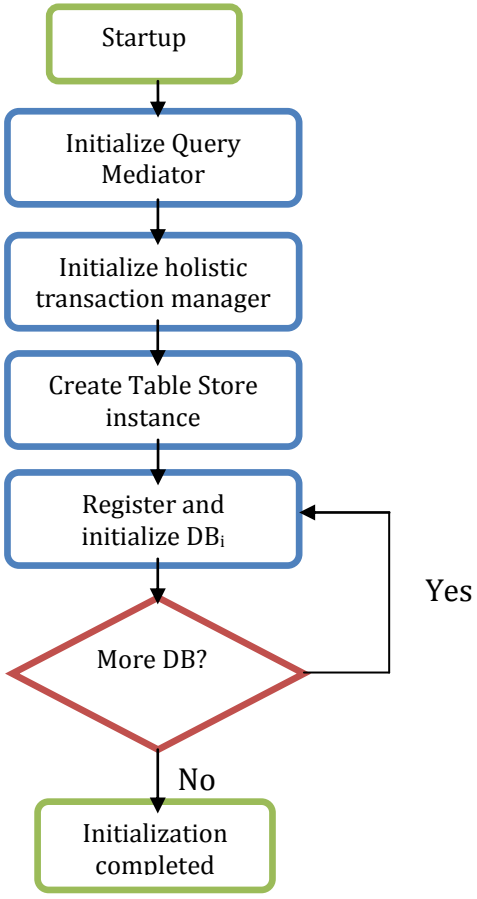

Compiler
Wrapper
JDBC
Query Mediator
Wrapper
Operator
Table Store connector
QUERY ENGINE 1
User Application
Load balancer
DB1
...
DBN
Table Store
QUERY ENGINE 2
QUERY ENGINE N

2.1. Query Engine Execution Flow

We describe the main execution flow of the query engine.

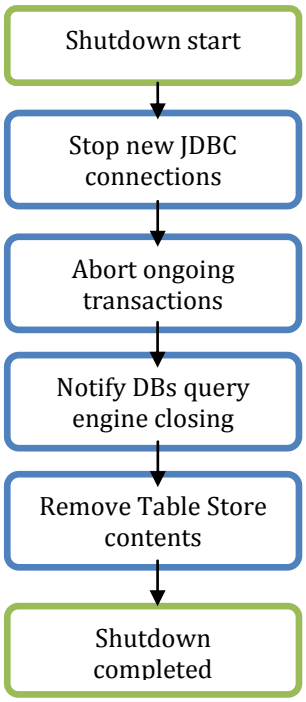
2.1.1. Startup

The startup procedure initializes the Table Store that will be used for that instance of the query engine, initializes the transaction manager and all databases available. The startup procedure will be used when no more instances of the query engine are already running.



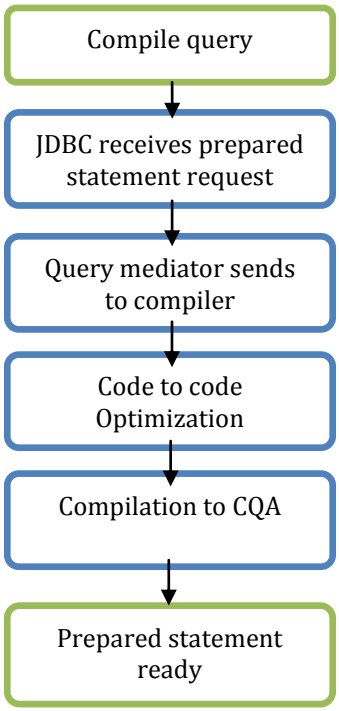
2.1.2. Shutdown

The shutdown procedure stops the arrival of new queries to the system and initiates the shutdown process. This process aborts pending queries, notifies all data stores about the shutdown and closes the Table Store.



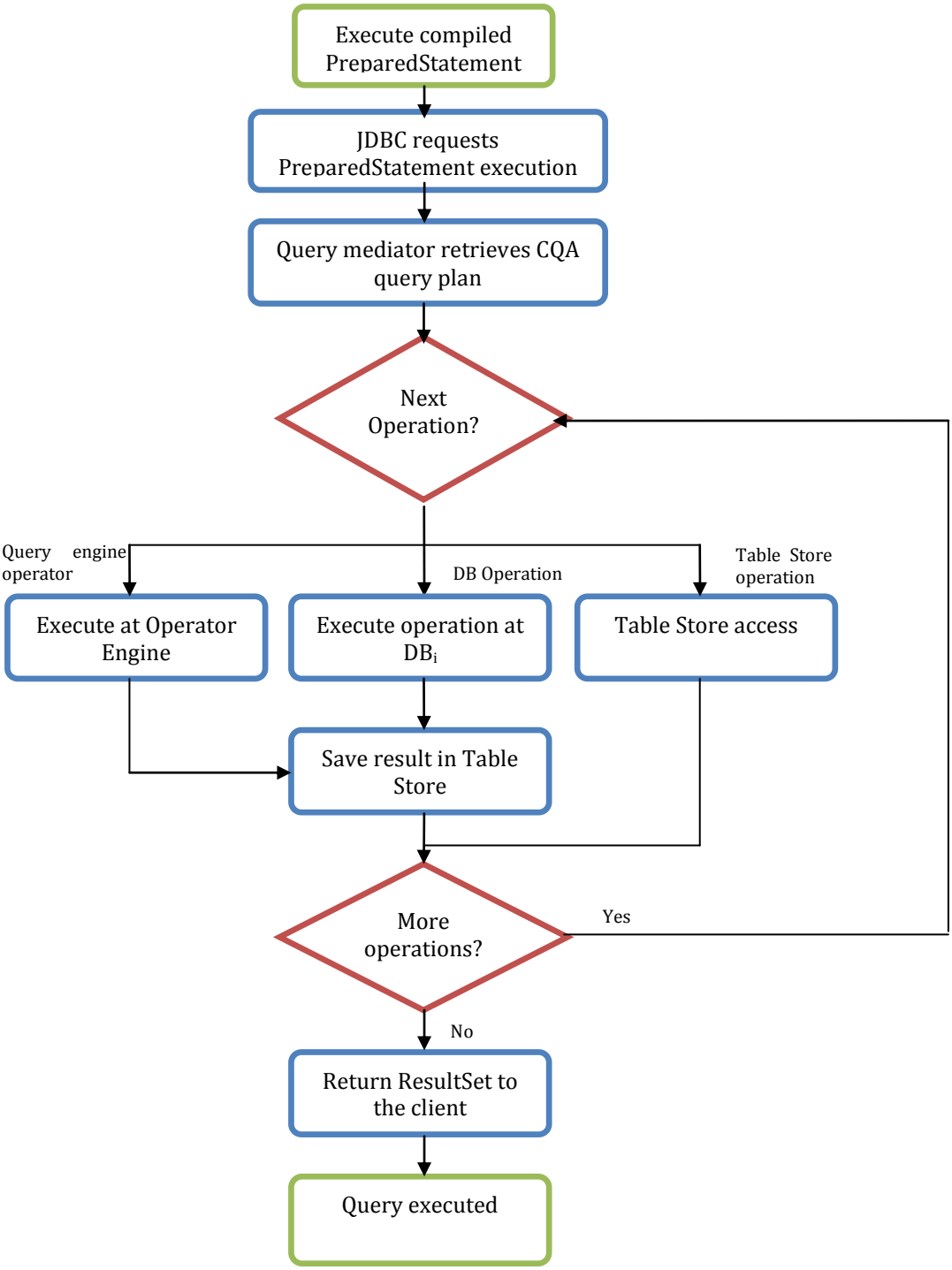
2.1.3. Query compilation

User code requests a PreparedStatement on the JDBC driver. This request is forwarded to the query compiler through the Query Mediator. The query compilation procedure optimizes and transforms the user query into a prepared statement that is ready to be executed.



2.1.4. Query execution

Client code starts a new query. The mediator retrieves the CQA precompiled query and interprets each operation: it checks the database engine that will perform the sub-query and forwards the operation to it.



2.1.5. Create and destroy instances of the distributed query engine

The query engine may have multiple replicas for scalability purposes as described in Section 2. In this mode of operation, the instances of the query engine follow a slightly modified version of the initialization and shutdown process when they are created or destroyed. The difference is that the table store is neither created nor destroyed by each individual instance, because the table store is shared among all instances. Even if an instance of the query engine is removed, the contents of the table store should remain visible for the other instances. Therefore, only the initialization of the first instance of the query engine creates the table store, and only the last one removes the contents of the table store.

2.2. Table Store

The query engine follows a stream model of operations, where the last operator pulls results from the previous one. The Table Store is the component that is used by the query engine to provide persistence to some temporal results and final results of the queries. Blocking operators that need to store large datasets will use the Table Store as the repository to save partial data. Temporal results may need to be consumed by multiple operators, which run at different speeds. When the fan-out operator (see Section 4.3) detects that the cache is not enough to deal with the operator speed mismatch, it starts storing the incoming elements in a temporary private table of the Table Store. Furthermore, final results that will remain accessible by name will also be materialized in the Table Store.

We define several requirements for the Table Store:

1. Store tabular data efficiently: the query engine represents all the data as tables. It is necessary that the Table Store represents the data as tuples and attributes in order to reduce the schema mismatch between the Table Store and the Operator Engine.
2. Storage of named tables: the Table Store needs to store multiple tables simultaneously and identify these tables by name.
3. Management of arbitrarily large data sets: queries among databases of the CoherentPaaS infrastructure may retrieve arbitrarily large datasets. Therefore, it is necessary that the Table Store has out of core capabilities to keep result sets that are larger than the memory available.
4. Easy access to data for the wrappers: each data provider in the project will be able to access the contents of the Table Store from its wrapper. Therefore, it is necessary that the Table Store provides easy interfaces to access the data.

We select MonetDB database as the storage of the Table Store. MonetDB fulfills the previously stated requirements as follows. First, it is a relational database management system and thus it is able to represent tabular datasets natively. Furthermore, MonetDB is a database that has shown excellent performance in benchmark evaluations due to its column store model and its implementation adapted to modern multicore CPUs. Second MonetDB allows the identification of the datasets by storing them in different tables using the relational model. Third, MonetDB offers an out of core functionality that is able to store large datasets and compress them on disk when the data does not fit in memory. Fourth, MonetDB has a fully compatible JDBC driver and an excellent integration with

Java, which is the implementation language of the Query Mediator. Therefore, MonetDB seems an adequate backend for the Table Store.

The Data Store operations provide a temporary storage where tables can be stored to be reused later by the current query or by others. Stored tables can be TEMPORARY only for the current session, or GLOBAL to be available for all sessions. Temporary tables are automatically destroyed when the session is finished, and global tables are removed by a drop command or when the engine is shutdown. In the following sections, we describe the operations to store and retrieve named tables from the Table Store.

2.2.1.1. SAVE(<source>, <name>, <options>)

It stores the result of a query operator as a table expression. The operator will pull rows from the source until it has produced all the results. The new table will be identified by the name given as the second parameter. The options parameter contains flags that modify the persistence of the table.

Input:

- `source` – **s**tores a table with a given name.
- `name` – name of the table
- `options` - optional property map with the options for the operator engine.
  - replace: TRUE to replace another table with the same name. When FALSE(default), it raises an error if the name already exists.
  - scope: GLOBAL or TEMPORARY (default)

Output:
If completed successfully, it returns a non empty table with one column and one row that indicates the number of rows stored. Otherwise, returns an empty table.

Examples:

```
SAVE(VALUES([INT],[[3],[2]]),'example',{'replace'=TRUE})
```

COL_1
2

2.2.1.2. LOAD(<name>)

Retrieves an stored table

Input:

- Name – name of the table

Output:
The contents of the stored table.

Examples:
```
LOAD('example')
```

COL_1
3
2

2.2.1.3. DROP(<name>)

Deletes a stored table.

Input:
- Name – name of the table

Output:
If completed successfully, it returns a non empty table with one column and one row with a value 1. Otherwise, returns a table which contains a 0.

Examples:
```
DROP('example')
```

COL_1
1

2.3. Query Engine Client: JDBC

From the perspective of a client that connects to the CoherentPaaS infrastructure, the query engine will be seen as a database. Therefore, we decided to implement the interaction between the clients and the query engine as a JDBC connection, which is one of the most popular database connection standards.

JDBC interface is part of the standard SDK and does not need to include additional libraries. We will create an implementation of the interfaces provided by java 1.7 SDK. We will implement the following interfaces:
- Connection: creates a connection between the client and the query engine.
- PreparedStatement: issues queries written in CloudMdsQL to the query engine. These queries will be compiled and later executed.
- ResultSet: provides a table like interface to retrieve the results from CoherentPaaS.

The choice of JDBC has several advantages that will facilitate the adoption of CoherentPaaS and an easy access, which will turn into a larger impact of the technology:
- Standardized access method: JDBC is part of the standard SDK, and thus it makes installation and testing easy. Additionally, database programmers are familiar to JDBC interfaces and can start using the CoherentPaaS infrastructure without learning a new database interface.

- Access from multiple environments: Java is a multiplatform language and implementations of the Java Virtual Machine run in almost any computing device: mobile phones, desktop stations and supercomputers. Furthermore, Java provides the Java Native Interface (JNI) that gives access to Java classes for any other language. Therefore, the CoherentPaaS infrastructure will be accessible to virtually any system.
- Reusability of older code: There is a large set of existing software applications that use JDBC interfaces to access the data. Since we will implement these interfaces, the adaptation of old code to CoherentPaaS will be easy because all the software logic can be kept. Programmers adapting old code will only need to update the queries issued to the database in order to take advantage of the CoherentPaaS cloud database

3. Query Compiler Architecture

The query compiler parses a CloudMdsQL query and generates a query execution plan that is executed by the query engine. It uses the Boost.Spirit framework for parsing context-free grammars, following the recursive descent approach. The compiler uses named table expression signatures or may query a catalog to retrieve metadata information necessary for semantic analysis. The execution plan is then delivered to the query processor in the form of a JSON document that contains a query execution tree with sufficient information to configure and run efficiently each of the query operations. The common JSON schema (according to draft 4 of http://json-schema.org) describing the format of a CloudMdsQL execution plan is provided in Appendix A. A valid execution plan must be able to be validated against this schema.

3.1. Query Decomposition

During query decomposition, the compiler builds the execution plan, which in its simplest form is a tree structure, where each non-leaf node represents a relational operation, and each leaf node represents a sub-query against a data store. At this stage, the compiler also prepares a set of native queries which will be passed to the corresponding wrappers and hence to the underlying data stores (this process will be explained later). Each node of the query plan tree represents a relational operation and an intermediate relation, result from the operation. Since the language allows the definition of named table expressions, which can be used as operands to several operations, it is possible that an intermediate relation is the input of more than one relational operator, therefore the query plan appears to be a graph rather than a tree structure.

While building the execution strategy, the compiler identifies a forest of sub-trees within the query plan, each of which is associated to a certain data store. Each of these sub-trees is meant to be delivered to the corresponding wrapper, who has to translate it to a native query and execute it against the data store. The rest of the query execution plan is the part that will be handled by the common query engine. Hence, we now outline two main subsets of the global execution plan:

- a forest of sub-trees that will be executed locally by each data store and
- a common query plan that will be executed by the common query engine, with leaf nodes consuming the relations returned by each wrapper as result of sub-tree execution.

At query decomposition stage, the boundary between the two subsets is preliminary and may be modified during the query optimization stage the following ways:

- the compiler may push selection operations from the common plan to sub-trees, in order to filter out as much data as possible before the query engine retrieves it, thus saving communication costs and improving the overall efficiency;
- the compiler may pull operations from sub-trees to the common plan if the corresponding data stores are not capable of performing them (see section 3.4).

3.2. Query Optimization

Due to the lack of global catalog and database statistics and because of the usage of native sub-queries that are considered as black boxes (and therefore cannot be analyzed), the capability of query engine to perform cost-based optimization is limited. Although some data stores can provide cost estimation of a particular sub-query, there are still many cases, for which the sub-queries are expressed in languages that cannot be subject to analysis, e.g. embedded Python code that invokes data store's API. For this reason, CloudMdsQL language gives its programmers possibility to specify execution directives by writing explicit join ordering and even explicitly specifying the join method to use. The compiler considers all explicitly specified JOIN…ON constructs in the FROM clause and thus identifies sub-trees from the common query plan that are considered atomic, i.e. not subject to join reordering. Besides taking into account explicit directives, the query optimizer always pushes selection operations down the query tree as much as possible, which may result in rewriting data store sub-queries by adding filter conditions.

To reduce the communication cost between data stores and the query engine, the latter can benefit from the usage of bind joins, which are efficient join operations between relations, retrieved from different data stores, processed following this approach: the left-hand side relation is retrieved, during which the tuples are stored in an intermediate storage and the distinct values of the join attribute(s) are kept in a list of values, which will be passed as a filter to the right-hand side sub-query. As an example, let us consider the following CloudMdsQL fragment:

```
SELECT a.x, b.y FROM b JOIN (BIND) a ON b.id = a.id
```

The execution directive `(BIND)` following the `JOIN` keyword means that join condition will be bound to the right-hand side of the join operation. First, the relation `B` is retrieved from the corresponding data store using its query mechanism. Then, the distinct values of `B.id` are used as a filter condition in the query that retrieves the relation `A` from its data store. Assuming that the distinct values of `B.id` are b_1 … b_n, the query to retrieve the right-hand side relation of the bind join uses the following SQL approach (or its equivalent according to the data store's query language):

```
SELECT a.id, a.x FROM a WHERE a.id IN (b1, …, bn)
```

Thus, only the rows from `A` that match the join criteria are retrieved. In order to perform this operation, the final sub-query to retrieve relation `A` must be composed by the query engine during runtime. Therefore, for each right-hand side of a bind join, the query compiler prepares an "almost ready" native query sentence, with placeholders for including the bind join condition, which will be added later by the query engine during runtime.

Whenever join ordering is not explicitly specified by execution directives, to generate a fairly efficient query execution plan at global level, the query optimizer may follow a best-effort approach according to the availability of local cost information that, whenever possible, should be provided by the wrappers. We introduce several approaches that can be adopted for the purpose. Each wrapper implementation may consider the cost-estimating capability of its data store in order to provide an efficient mechanism to perform cost estimation of sub-queries. Any of the described below

methods can be used for the purpose, depending on the features provided by the data store:

- If the data store can efficiently estimate the cost of a sub-query and the size of its result set (like EXPLAIN on prepared statements), the query engine may benefit from this to directly estimate the cost of a sub-query.
- In other cases, the wrapper implementation can make use of available database statistics (like cardinalities, number of distinct values per column, etc.) and metadata information (like the availability and types of indexes) in order to provide cost and selectivity functions to support the global optimization task.
- If none of the above methods are applicable, but the data store can process aggregate queries like COUNT(*), MIN and MAX, the wrapper can periodically run in background such queries, thus synthesizing and keeping statistics like the number of tuples in a table or the min/max values of an attribute.

However, the lack of cost models in some NoSQL data stores and the limited (or lack of) capability to build database statistics do not allow for defining a precise global cost model for query optimization. Therefore, the solution to the optimization problem has to deal with incomplete cost information following approaches involving heuristics, dynamic query optimization and user-defined cost and selectivity functions.

Sub-query rewriting can be planned by the optimizer in several occasions:

- selection pushdowns which result in pushing filter conditions from the common plan to sub-trees;
- usage of bind joins which implies adding filter conditions to the sub-query in order to allow the retrieval of only those tuples that match the join criteria;
- taking advantage of sort-merge joins which requires adding sorting operations to sub-queries in order to guarantee that the retrieved relations are sorted by their join attributes.

The first rewriting approach is considered always efficient, i.e. whenever the data store is capable of handling it, the optimizer will plan selection pushdown. However, bind joins or merge joins will be planned either if explicitly specified by CloudMdsQL directives or as a result of optimization decision, of course taking into account data store's capabilities as well.

3.3. Sub-querying SQL Compatible NoSQL Data Stores

Since the data model of some NoSQL data stores (e.g. key-value or document databases) can be considered as a subset of the relational model, in most cases it is possible to map simple SQL commands to native queries, without compromising the functionality. In fact, SQL-like languages are already commonly used with data stores based on the BigTable data model, e.g. CQL for Cassandra. For such data stores, the recommended approach for sub-querying within CloudMdsQL is to use SQL table expressions against the data store, even though the data store does not natively support SQL. 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 rewritten by the optimizer before submitted for execution to the data store. This allows the CloudMdsQL engine to perform optimizations of the global query execution plan (like pushing selections, projections and join operations down the tree as much as possible) or take advantage of bind joins, etc. Each sub-tree is then delivered to the

corresponding wrapper, which interprets and transforms it to a native query, in order to execute it against the data store using its native query mechanism.

3.4. SQL Capabilities

In order to build executable sub-query plans, the query engine must be aware of the capabilities of the corresponding data store, to perform operations supported by the common data model. Therefore, the wrapper implementer must identify the subset of the common algebra that is supported by the data store. Thus, the query planner can decide on which parts of the global query plan can be handled locally by the data stores and which part should remain in the common query plan (see Section 3.1). For example, a MongoDB data store can perform selection operations – analogous to the document collection method find() – but is not able to perform joins. Being aware of that, the query planner can push selection operations down to the sub-query plan, but will assign any join operation between MongoDB document collections to the common query plan.

The method to handle data source capabilities, proposed in [Tomasic98], requires that the query engine serializes the sub-query plan (or single operations from it) to a sentence of a specific language, that should be matched against a pattern, provided by the corresponding wrapper – if the validation succeeds, then the data store is capable of executing the sub-query. Thus the query planner can determine the boundary between the common query plan and the sub-plan that will be handled by the data store.

In CloudMdsQL a similar approach is proposed which makes use of JSON schemas as an instrument for the wrapper to express its data store's capabilities. To test the executability of a sub-plan (or a single operation) against a data store, the query planner serializes it to a JSON document that has to be validated against the JSON schema exposed by the wrapper. Below is an example of a capability JSON schema for a key-value data store, that is capable only of performing selection operations involving comparisons on the 'key' attribute (only certain elements of the schema object are shown):

```
{
  "properties": {
    "op": { "type": "string", "pattern": "SELECT" },
    "tableref": { "type": "string" },
    "filter": { "$ref": "#/definitions/expression" }
  },
  "definitions": {
    "expression": { "oneOf": [
      { "$ref": "#/definitions/comparison" },
      { "$ref": "#/definitions/function" }
    ] },
    "comparison": { "properties": {
      "comp": { "type": "string", "pattern": "=|<|>|<=|>=|<>" },
      "lhs": { "properties": {
        "colref": { "type": "string", "pattern": "key" },
      },
      "rhs": { "type": "string" }
    } },
    "function": { "properties": {
      "func": { "type": "string", "pattern": "AND|OR" },
      "lhs": { "$ref": "#/definitions/expression" },
      "rhs": { "$ref": "#/definitions/expression" }
```

```
    } }
  }
}
```

Now let us consider the following sub-query that is composed of two conjunctive selection conditions, each of which is tested against the capability specification. The result of the validation shows that condition #1 can be handled by a selection operation in the key-value data store and therefore it will be left in the sub-query, while condition #2 doesn't pass the validation, and therefore will be pulled up in the common plan to be processed by to common query engine.

```
SELECT key, value FROM tbl WHERE key BETWEEN 10 AND 20 AND value > key
```

Condition #1: `key BETWEEN 10 AND 20` Validation: success	Condition #2: `value > key` Validation: failure

`{`
`  "op": "SELECT",`
`  "tableref": "tbl",`
`  "filter": {`
`  "func": "AND",`
`  "lhs": { "comp": ">=",`
`  "lhs": {"colref": "key"},`
`  "rhs": "10" },`
`  "rhs": { "comp": "<=",`
`  "lhs": {"colref": "key"},`
`  "rhs": "20" }`
`  }`
`}`
	`{`
	`  "op": "SELECT",`
	`  "tableref": "tbl",`
	`  "filter": {`
	`  "comp": ">",`
	`  "lhs": {"colref": "value"},`
	`  "rhs": {"colref": "key"}`
	`  }`
	`}`

Generally, for each data store it is important to identify the intersection of the set of its supported operators and the set of CloudMdsQL operators. Therefore, a common procedure is, for each wrapper implementer, to outline a subset of the common JSON schema (see Appendix A) in order to prepare the capabilities schema for its data store.

4. Query Operator Engine Architecture

The query operator engine is the component that combines the results from the data stores by performing classical relational operations (selection, join, grouping, etc.). The operators will be implemented following an iterator (Volcano-like) model [Graefe94] with highly efficient operators optimized for SMP architectures, taking advantage of the multiple cores and threads available, and being conscious of the different memory cache levels.

Each of the query operators receive one or more cursors of tabular data and optionally some configuration parameters such as filter conditions, sorting order, etc. Each cursor contains one or more columns of data, where each column is restricted to a single data type of a fixed length binary representation (e.g. integer, double precision or timestamp), or a fixed length offset to a dictionary of variable length values such as UNICODE strings. Thus, chunks of data can be efficiently managed as a vector of fixed length values by taking advantage of SIMD instructions as presented in [Boncz05]. Columns are also split into chunks of data of a small size, usually a multiple of the operating system page size (a few Kbytes).

The iterator model processes the query from top to bottom: at each step one chunk of data for each column is provided to a consumer operator. Pipelined chunks are never modified and the only updated chunks are those that denote when a value for a column of a row has been set to NULL. By using this approach, chunks have a higher probability to stay in the inner CPU caches and the cache coherence is easier to maintain, which results in an improved performance and a better usage of the multicore capabilities in a SMP architecture. This iterator approach obtains the result tuples as soon as they are generated, unless there is a blocking operator such as grouping or sorting. This final table is retrieved by the application with a forward sequential tuple iterator that supports rewinding and repositioning into marked rows. When the result table is no longer required then it is automatically removed from the temporary storage.

The query operator engine will be able to execute CloudMds Query Algebra (CQA) programs, which are sequences of operators expressed in the JSON format described in Section 3. CQA programs chain the operators following the Volcano model by means of iterators on streams. Therefore, the construction of sequences of operations will make that the operators are not interpreted line by line from the JSON plan. Also, the direct flow of data among operators avoids dumping back and forth all the results to the table store.

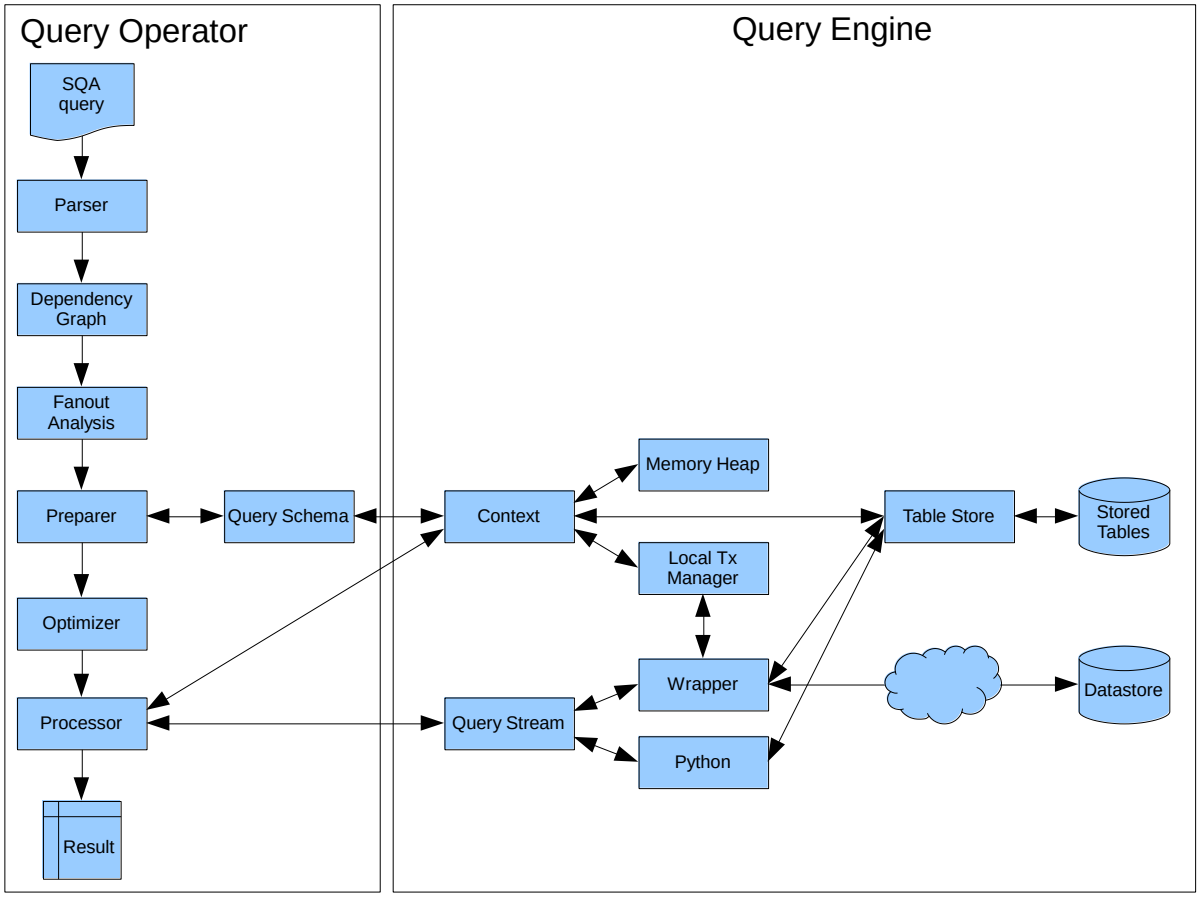
The query operator engine will be based on the new graph query operator engine that Sparsity is building for Sparksee, which implements the previously described architecture. They will share the parsing, query plan analysis and preparer. The query operator engine will take also advantage of the multithreading capabilities that Sparsity is implementing in the new graph query operator engine. But, there are several important differences between the operator engine of CoherentPaaS and the one of Sparksee:

- The graph query operator engine is based on graph specific operations such as getting neighbors or accessing edge data. On the other hand, the query operators

for CoherentPaaS are relational operators. At the end of the project, Sparsity will evaluate, as an exploitation of the project, if providing relational operators in Sparksee is useful for graph use cases.

- The operator engine will not depend on Sparksee and will not need a Sparksee database instance to be deployed. The backend to store data of the operator engine is the Table Store module of CoherentPaaS. On the other hand, the graph query operator uses Sparksee data structures to provide persistence and store temporary results.
- The query operator engine will not need a Sparksee license. However, the graph query operator is part of Sparksee and will need a license.

The query engine is composed of the modules shown in the following figure:



4.1. Parser

The *parser* component parses the JSON query and generates a *dependency graph*. It is a LALR(1) parser implemented with the *lemon*[1] tool. Each instance of the parser reads the tokens using a *tokenizer*, and tokenizers get the text from string *readers*.

As the parser reads the query text, it generates a *runop* instance for each operator. All runops belong to a single dependency graph. The result of a query is the result of its topmost operator, which is also the root of the dependency graph. Examples of runops are SELECT, JOIN, PROJECT, etc.

[1] http://www.hwaci.com/sw/lemon/

4.2. Dependency Graphs

In the current version, there is a main dependency graph per query and each subquery has its own dependency graph which is a child of the main one.

Each runop in a dependency graph has a list of zero or more producers, its arguments, and a list of zero or more consumers. Only the root operator has zero consumers and, when an operator has more than one consumer, then it is implicitly a *fan-out candidate*.

Operators can have also scalar parameters (numbers, strings, etc.), collections such as lists or property maps, or even expressions.

4.3. Fan-out Analysis

After the syntax phase that reads the query and generates the dependency graph, all operators are scanned to detect fan-outs (more than one consumer). Each operator with fan-out is replaced in the dependency graph by an instance of a fan-out special operator. This operator has one single producer (the original fanned out operator) and as many consumers as his producer had.

After this substitution, only instances of fan-out operators will have more than one consumer. In runtime, each fan-out operator will cache its input data to provide virtual copies to its consumers. When no more consumers are active, the cached data will be released. If the data set becomes too large to be stored in the cache of the fan out operator, the operator will create a temporary table in the Table Store. When the operator finishes the execution, the temporary table will be deleted.

4.4. Preparer and Query Schemas

The next step is the semantic validation. Starting from the root in a top-down approach, each operator is invoked to be *prepared* for execution, or, in other words, to check if their inputs are valid. Also, during this process each operator figures out the structure of its output (number of columns and data types) from its capabilities, parameters and inputs.

Also, each expression parameter is also compiled into a *compiled expression*, which is a prefix representation of the expression, and much more easy to evaluate using a stack. During this compilation there is also a verification of the semantics of the expression, and an optimization to simplify it to a more efficient and equivalent expression.

4.5. Optimizers

The optimization procedure follows a cascade model. Once the dependency graph has been validated, one or more *optimizers* are invoked in sequence. Each optimizer receives a dependency graph, and returns a dependency graph with the only restriction that it must compute the same result. The returning graph is either the same graph when the current optimizer could not apply any optimization, or an equivalent dependency graph with an expected lower computational cost. Some examples of optimizers are CSE (common sub-expressions), filter push-down, etc.

At this point, the query has been fully prepared and it is ready for execution.

4.6. Processor

A prepared query can be executed one or more times. The current query processor only allows one concurrent execution of a prepared query, and it is single threaded. Memory buffers for the operators are provided by a query *context*.

As stated before, the processor is a pull model with a pipeline of chunks of rows. Each runop creates a single *cursor* that will be pipelined. The producers of a runop provides their cursors as arguments to their consumers.

The processor starts by executing the root runop. For each operator, first it executes its parameters or, in other words, gets their cursor. Then it creates the operator cursor. The pipelining process consists in the classical pull approach where each cursor *consumes* data from its arguments, in a cascade approach until the lowest operators in the tree, which are the access methods.

At each invocation of a *fetch* method of a cursor, it returns a chunk of rows. The chunk size is obtained in runtime from the context, and it is usually a multiple of the operating system page size. Also, instead of managing rows, cursors have a column-store approach were each column is a vector of values. For fixed size data types, each element of the vector is a value, while for variable-length data types, such as strings, there is an auxiliary pool of texts.

Vectors are never modified. If a chunk contains deleted rows, there is an auxiliary buffer that informs of which rows are valid. Also, null values are denoted with a specific buffer with a bit mask to mark nulls. The current limit is 64 columns because the presence bits are stored in a 64 long value.

Thus, a column buffer is only marked as dirty when it is created. After that it is only accessed in read-only mode. And, if the column needs to be modified, then a new buffer is created to improve the cache usage. Only the buffers for deleted rows, null presence and variable-length pools can be modified between operators.

With this pipelining approach, most of the operators are non-blocking and data is generated from bottom to top in small chunks. There are some blocking operators, such

as aggregates or joins, that cannot return the first chunk until some of their input have been fully processed.

Finally, the results of the query are the chunks generated by the cursor of the root runop. This data is cached (as for fan-outs), until the application closes the iterator.

4.7. CQA Programming Interface

This section explains how to use the CQA using the following new classes:

- `Query` : a query executed inside a `QueryContext`
- `ResultSet` : the result of a query execution
- `QueryStream` : interface to allow the application to provide data during the query execution
- `QueryContext` : application-defined environment to store temporary data. In CoherentPaaS, the QueryContext will map to the Table Store.

4.7.1. Query

A `Query` in CQA encapsulates a program that can be executed inside the scope of a `QueryContext`, and returns a single `ResultSet` composed by a collection of rows and columns. A simple example of the usage of a `Query` is:

```
Value v = new Value();
QueryContext ctx = new MyAppQueryContext();
Query q = ctx.newQuery();
ResultSet rs = q.execute("VALUES([INT],[[1],[2],[3]])");
System.out.println(rs.getColumnName(0) + "=====");
while (rs.next()) {
    rs.getColumn(0, v);
    System.out.println(v);
}
rs.close();
q.close();
```

which prints out the query result:

COL_1
1
2
3

The content of a query can also be exported in `JSON` documents. For example:

```
Value v = new Value();
QueryContext ctx = new MyAppQueryContext();
Query q = ctx.newQuery();
ResultSet rs = q.execute("VALUES([INT],[[1],[2],[3]])");
System.out.println(rs.getJSON(2));
System.out.println(rs.getJSON(1));
```

```
rs.close();
q.close();
```

splits the result in two `JSON` documents:

```
{
 "columns": [
  { "COL_1": "INT" }
 ],
 "rows": [
    [ 1 ],
    [ 2 ]
 ]

"columns": [
  { "COL_1": "INT" }
 ],
 "rows": [
    [ 3 ]
 ]
}
```

The `execute` method is solved in two phases: the *prepare* step, where the query is compiled and verified, and the *run* step, where the query is evaluated and solved. This is known as the *prepared statement* procedure that will be fully supported in future versions. Between both phases, the query can embed values inside placeholders for dynamic parameters that are required for the query execution. An example of dynamic parameter usage is:

```
Value v = new Value();
QueryContext ctx = new MyAppQueryContext();
Query q = ctx.newQuery();
v.setString("demo");
q.setDynamic("text", v);
try
{
    ResultSet rs = q.execute("VALUES([STRING],[[$text]])");
    rs.next();
    rs.getColumn(0, v);
    System.out.println("$" + v + "$");
    rs.close();
} catch (RuntimeException qe) {
    System.out.println("Query Exception catched: " + qe.getMessage());
}
q.close();
```

which prints out:

```
$demo$
```

Note that in this last example, both JSON parsing errors and execution errors are caught as exceptions.

4.7.2. Streams

The query is executed as a whole by the query engine. This reduces the amount of communication between the application and the engine through the APIs, with an increase of performance due to the removal of data transformations in the wrapper, the reduction of the checks for valid parameters, and by potential optimizations provided by the query engine. But, in some cases, it is necessary to allow the application to interact with the query execution, for example to solve an algorithm that is not supported in CQA, or to provide data extracted from external data sources. The mechanism that supports this in CQA is known as *streams*. A CQA `STREAM` is an application-defined operation that receives zero or more arguments (scalar values or temporary tables), and returns a stream of rows and columns.

Streams are managed through a `QueryStream` instance. Each subclass of `QueryStream` must implement three methods:

- `prepare` - before the execution of the stream, receives the scalar values, and prepares the instance for the execution
- `start` - the query engine has obtained the first rows of the temporary tables that are parameters of the stream
- `fetch` - the query engine requests another row from the stream

The following example shows a stream that multiplies an input collection of numbers by a fixed multiplier.

```
class StreamMult extends QueryStream {
    private double mult;
    private ResultSet res;

    // [in] args - input list of values
    // returns false on error
    public boolean prepare(ValueList args) {
        if (args.count() != 1) {
            return false;
        }

        Value v = args.get(0);
        if (v.getDataType() != DataType.Double) {
            return false;
        }

        mult = v.getDouble();
        return true;
    }

    // [in] input - optional list of input ResultSets
    // returns false on error
    public boolean start(ResultSetList input) {
        if (input.count() != 1) {
            return false;
        }

        res = input.get(0);
        if (res.getNumColumns() != 1) {
            return false;
```

```
        }
        if (res.getColumnDataType(0) != DataType.Long) {
            return false;
        }
        return true;
    }

    // [out] next row, or empty for end of iterator
    // returns false on error
    public boolean fetch(ValueList result) {
        if (res.next()) {
            Value v = new Value();
            res.getColumn(0, v);
            if (v.getDataType() == DataType.Long) {
                double d = v.getLong();
                result.add(v);
                v.setDouble(d * mult);
                result.add(v);
                return true;
            } else {
                return false;
            }
        }
        return true;
    }
}

Value v = new Value();
QueryContext ctx = new MyAppQueryContext();
Query q = ctx.newQuery();
StreamMult sMult = new StreamMult();
try
{
    q.setStream("mult", sMult);
    v.setLong(3);
    q.setDynamic("value", v);
    rs = q.execute("STREAM('mult',[LONG,DOUBLE],[2.5],
                   [ VALUES([LONG],[[$value]]) ])");
    System.out.println(rs.getJSON(10));
    rs.close();
} catch (RuntimeException qe) {
    System.out.println("Query Exception catched: " + qe.getMessage());
}
q.close();
```

which returns:

```
{
 "columns": [
  { "COL_1": "LONG" },
  { "COL_2": "DOUBLE" }
 ],
 "rows": [
    [ 3, 7.5 ]
 ]
}
```

4.7.3. QueryContext

A query is executed inside a `QueryContext`, which is an interface that provides all the resources required by the query engine, such as:

- Memory management for query buffers.
- Access to external resources (files, data streams, ...).
- An interface to access a store, which materializes the results.

The current implementation only encapsulates a very simple memory heap, and raises an error for any other operation. In future versions, the interface will allow subclasses to implement all the required methods and connect to the Table Store.

4.8. CQA programs

The CloudMds Query Algebra (CQA) allows for the creation and execution of programs (expressions) for the resolution of queries. CQA expressions are solved by combining classical relational operators (selection, join, grouping, etc.) and calls to custom functions written in native code through the stream operator.

Each of the CQA operators receive one or more cursors of data and, optionally, some configuration parameters such as filter conditions, sorting order, etc. Each cursor contains one or more columns of data, where each column is restricted to a single data type of a fixed length binary representation (e.g. integer, double precision or timestamp), or a fixed length offset to a dictionary of variable length values such as UNICODE strings.

A CQA query returns a single table or result set, in the form of rows and columns that can be iterated or formatted in a JSON document.

The following subsections describe in detail the CQA syntax.

4.8.1. Comments

There are two types of comments: *line* comments and *block* comments.

A *line* comment ignores all the text from the marker until the end of the line. The line comment styles are:

- SQL line comment: two dash characters '–'
- SPARQL line comment: one hash character '#'
- C++ and Cypher line comment: two slash characters '/'

A *block* comment ignores all the text between the start marker `/*` and the end marker `*/`. Both markers are required: if the end marker does not exist, the parser raises a syntax error. Block comments cannot be nested and line comments are ignored inside a block comment.

Some examples of comments are:

```
-- SPARQL-style comment
# SQL-style comment
// C++ style line comment
/* here a block comment starts
 everything is ignored
 -- even this line comment
 up to the end marker
 */
```

4.8.2. Reserved Words

The *reserved words* are those words with a specific meaning into the CQA algebra. Reserved words are *case insensitive*: they can be written in any combination of uppercase or lowercase letters.

The current list of reserved words is:

```
ADJACENT ALTER_ATTRIBUTE ALTER_TYPE AND ATTRIBUTES BETWEEN BOOL BOTH
CONNECT CREATE_ATTRIBUTE CREATE_EDGE_TYPE CREATE_INDEX_ATTRIBUTE
CREATE_NODE_TYPE DEFAULT DOUBLE DROP_ATTRIBUTE DROP_TYPE EDGES EXCEPT FALSE
GET GLOBAL GRAPH IN INDICES INGOING INSERT_EDGES INSERT_NODES INT LET LONG
MATCH NEIGHBORS NOT NULL OR OUTGOING PROJECT REMOVE RENAME SCAN SELECT
SEQUENCE SET SLICE STATISTICS STREAM STRING TIMESTAMP TRUE TYPES UNIQUE
VALUES
```

4.8.3. Identifiers

An *identifier* is a sequence of latin characters, digits and underscore characters that starts by a latin character or an underscore. Identifiers are *case sensitive*, and at least one character must be latin. Some examples of distinct identifiers are:

```
alfa Alfa ALFA Name_45 _A
```

Instead, the following are not valid regular identifiers:

```
45Name álfa _ __ _45
```

4.8.4. Values

CQA supports five different kinds of values: constants, undefined (null), lists, property maps, and dynamic parameters.

4.8.4.1. Data types and constants

The basic data types supported in CQA are:

- `INT`: signed 32-bit integer numbers
- `LONG`: signed 64-bit integer numbers with the suffix '`L`'
- `DOUBLE`: IEEE-754 double precision floating-point numbers
- `BOOL`: logic value `TRUE` or `FALSE`
- `TIMESTAMP`: ISO-8601 date and time with millisecond precision and UTC time zone offsets. Timestamp values are surrounded by ampersand delimiters. The valid range is from &1970-01-01T00:00:01Z& to &2038-01-19T03:14:07Z&.
- `STRING`: sequences of UNICODE characters delimited by single quotes. Special characters can be escaped by the backslash character '\':
  - '`\'`' : a single quote
  - '`\\`' : backslash
  - '`\b`' : backspace
  - '`\t`' : horizontal tab
  - '`\n`' : newline
  - '`\f`': form feed
  - '`\r`' : carriage return

Some examples of valid constants are:

```
-32456 123789
3435973836800L -1L
64.3 -0.2E-4
'O\'Hara' 'Three\n\tÁÈü\nUNICODE lines'
TRUE FALSE
&2010-10-21& &1999-12-31T23:59:59.999Z& &2010-10-21Z-02:15&
```

4.8.4.2. Undefined (NULL) Values

`NULL` denotes an undefined value. Any operation with a `NULL` operand always returns `NULL`, and two `NULL` values are distinct because they are undefined. The behavior of nulls will match the definition of null in CloudMdsQL.

4.8.4.3. Lists

A *list* is an ordered collection of values. Lists surrounded by square brackets, and the values separated by commas. The values can be of different data types. Some examples are:

```
[] -- empty list
[1, 2, 3] # sequence of numbers
['ABC', NULL, True] // different value types
```

4.8.4.4. Property Maps

A *property map* is a collection of properties. Each *property* is identified by a distinct name and associated to a value. Property maps are surrounded by curly brackets and each property is identified by a `STRING` literal. When a property is defined more than once inside a map, only the last appearance is taken into account. Some examples are:

```
{} -- empty property map
{ 'a'=1, 'b'=FALSE, 'c'=NULL }
{'x'=1.0,'x'=2.0} -- single property 'x' with value 2.0
{'list'=[1,2]} -- list as property value
{'map'={}} -- map as property value
```

4.8.4.5. Dynamic Parameters

A *dynamic parameter* is a constant value that will be supplied by the application before the execution of the query. A query can have multiple dynamic parameters, each one identified by a different identifier prefixed by the '@' character.

A dynamic parameter has no data type, and its expected data type is deduced by the parser from the semantics of the query. The query engine expects that a valid value for any dynamic parameter will be set on runtime.

Valid examples of dynamic parameters are:

```
@A
@ B45
@Yes_No
```

Lists and property maps are not supported as runtime values in the current version of CQA.

4.8.5. Expressions

An CQA expression is a combination of values, operators and functions that are interpreted according to particular rules of precedence and of association. When evaluated inside a query, an expression produces a single result value.

Parentheses can be used to explicitly denote precedence, by grouping parts of the expression that should be evaluated first.

The current version of CQA supports only values (constants, NULL, lists, property maps and dynamic parameters). The following list, in precedence order, describes the operators and functions:

- `X AND Y`: logical AND between expressions X and Y. It is a short-circuit (lazy) operator that only evaluates Y when X evaluates to TRUE.
- `X OR Y`: logical OR between expressions X and Y. It is a short-circuit (lazy) operator that only evaluates Y when X evaluates to FALSE
- `NOT X` : logical NOT of expression X
- `=, <>, <, <=, >, >=` : comparison operators between two expressions
- `+, -` : binary addition and subtraction operators
- `*, /` : binary multiplication and division operators
- `+, -` : unary sign operator
- `%S` : returns the value in the column named as S

- `%n` : returns the value in the column with index n >= 0
- `BETWEEN(V,L,H)` : returns TRUE when L <= V <= H
- `IS_NULL(V)` : returns TRUE if V evaluates to NULL
- `IN(V,L)` : returns TRUE if the result of V is inside the list L
- `MATCH(V,P,M)` : returns TRUE if the result of V matches the pattern P. The property map M is used to specify the escape character, case insensitive matching, etc.
- `datatype(V)` : converts the result of V to another datatype which can be INT, LONG, DOUBLE, STRING, BOOL or TIMESTAMP
- `CASE(V,[`C_1`:`V_1`,...`C_n`:`R_n`],D)` : checks if the result of V is equal to any C_i: if so, returns its V_i, otherwise it returns D

4.8.6. CQA Queries

CQA queries are expressed as a combination of relational. The arguments of an operator can be lists, property maps, or the result of the evaluation of expressions or other CQA operators. An operator always returns a single table. The final result of an CQA expression is always the result of its root operator.

For example:

```
RENAME(SLICE(VALUES([INT],[[1],[2],[3]]), NULL, 1), ['top-1'])
```

- has three nested operators: `RENAME`, `SLICE` and `VALUES`
- `RENAME` is the root operator
- the result of `RENAME` is the result of the expression
- the result of `VALUES` is an argument for `SLICE`

More complex programs can be written with the `LET` construct. A `LET` is a sequence of one or more CQA expressions that are evaluated and their results are stored in temporal variables. Variable names are regular identifiers followed by the character '`@`'. When a variable is defined, the following operations can reference to this variable by such name. The previous example can be represented using `LET` as:

```
LET @values = VALUES([INT],[[1],[2],[3]]),
    @top = SLICE(@values, NULL, 1)
 IN RENAME(@top, ['top-1'])
```

Both expressions are equivalent. Notice that expressions are evaluated in sequence: first `VALUES`, then `SLICE`, and finally `RENAME`.

4.9. CQA operators

4.9.1. Basic Operations

The basic operations construct tables or perform simple transformations over a table.

4.9.1.1. VALUES(<datatypes>, <rows>)

Returns a new table populated with a predefined set of rows and column values. The value for each column of each row must match the expected data type. Each row cannot have a number of values greater than the number of columns. Rows with fewer values than columns are filled with NULL values.

Input:

- `datatypes` - non-empty list of datatype: INT, STRING, etc.
- `rows` - optional list of rows, where each row is an optional list of values

Output:

A new table with one column for each element in `datatypes`, and filled with `rows`.

Examples:

```
VALUES([STRING],NULL)
```

COL_1

```
VALUES([STRING,INT,LONG,BOOL,DOUBLE,TIMESTAMP],
       [null,
        [],
        [null, 32],
        ['text', null, null, TRUE, 12.7, &2014-05-28&],
        ['hello\nworld', -12, 87888L, FALSE, 0.5e-2,
         &2013-01-2T22:15:18.321+01:00&]])
```

COL_1	COL_2	COL_3	COL_4	COL_5	COL_6
	32				
text			TRUE	12.7	2014-05-28T00:00:00.000Z
hello	-12	87888	FALSE	0.005	2013-01-12T23:15:18.321Z

4.9.1.2. SLICE(<source>, <skip>, <limit>)

Returns a subset of the input rows.

Input:

- `source` - input table
- `skip` - optional, number of rows to skip from the beginning (NULL for none)
- `limit` - optional, maximum number of rows to return (NULL for all)

Output:

The input `source` without the first `skip` rows (optionally) and up to a maximum of `limit` rows (also optional).

Examples:

```
SLICE(VALUES([INT],[[1],[2],[3]]), 1, NULL) -- all except the first one
```

COL_1
2
3

```
SLICE(VALUES([INT],[[1],[2],[3]]), NULL, 2) -- top 2
```

COL_1
1
2

4.9.1.3. RENAME(<source>, <names>)

Changes the names of one or more columns.

Input:

- `source` - input table
- `names` - list of new column names. Each name must be a *string* or NULL to keep the original name. The length of the list must be the same as the number of columns in `source`

Output:

The input `source` with some column names renamed.

Examples:

```
RENAME(VALUES([INT, DOUBLE],[[1, 3.14]]), [NULL, 'Pi'])
```

COL_1	Pi
1	3.14

4.9.1.4. PROJECT(<source>, <columns>)

Returns a subset of the columns in `source`. The order of the columns is the same as their appearance in the list, and a column can appear only once one.

Input:

- `source` - input table with N columns
- `columns` - list of columns. Each column is identified by an *int* number in the range from 0 to N-1

Output:

A subset of columns of all the rows in `source`.

`Examples:`

```
PROJECT(VALUES([INT, LONG, BOOL],[[1, 2L, TRUE]]), [2, 0])
```

COL_3	COL_1
TRUE	1

4.9.1.5. STREAM(<stream>, <columns>, <parameters>, <input>)

Executes an application-defined operation. The operation receives some parameters and input tables, and returns a stream of rows.

Input:

- `stream` - application-defined name of the operation
- `columns` - list of data types of each column expected in the result
- `parameters` - optional list of values used to initialize the operation
- `input` - optional list of input tables to be used by the operation

Output:

As many columns as defined in `columns`, and the rows are generated as a stream by the application.

Examples:

```
STREAM('mult-div',[INT,DOUBLE,DOUBLE],[2.5,2.0],
       [ VALUES([INT],[[1],[2],[3]]) ])
```

COL_1	COL_2	COL_3
1	2.5	0.5
2	5	1
3	7.5	1.5

4.9.1.6. SEQUENCE(<op$_1$>, ..., <op$_n$>)

Executes in sequence all the operations, and returns the result of the execution of the last operation. The intermediate results are discarded. This operation is useful when it is necessary to execute independent operations in a specific order.

Input:

- op_1, ..., op_n - operations to execute

Output:

The result of the execution of op_n.

Examples:

```
SEQUENCE(VALUES([INT], NULL), SLICE(VALUES([INT],[[1],[2],[3]]), 1, 1))
```

COL_1
2

4.9.1.7. READ(<filename>, <columns>, <options>)

Reads a CSV (column separated value) data file. The file can be optionally compressed in GZ format.

Input:

- `filename` - *string*, name of the CSV file
- `columns` - list of pairs [`datatype props`] with one entry for each column in the file, where:
  - `datatype` : datatype of the column
  - `props` : optional property map
    - `skip`: *bool*, to ignore this column in the output
  - `options` - property map to configure the input

- `locale` : *string*, locale (default "`en_US`")
- `separator` : *string*, column separator character (default `','`)
- `quote` : *string*, string delimiter (default `'"'`)
- `multiline` : *int*, 0 for single line values, or the maximum number of rows for a multiline value (default 0)

Output:

One row for each row in the data file, with one column for each column in the file except those `skipped`.

Examples:

```
READ('movies.txt.gz',[INT {'skip'=true}, STRING, INT],{'separator'='|'})
```

COL_1	COL_2
A-Team	2006
Access denied	2005
Acid Rain	2006

4.9.2. Data Operations

The data operations are those that modify the contents of the rows, generate calculated data, or operate between two or more inputs.

4.9.2.1. SELECT(<input>, <filter>)

Returns the rows that evaluate to true for a given condition.

Input:

- `input` - input table
- `filter` - a boolean expression

Output:

The filtered rows with the same columns as the input.

Examples:

```
SELECT(VALUES([INT,STRING],[[1,'A'],[2,'A'],[3,'B']]),
       BETWEEN(%0,1,2) AND %1='A')
```

COL_1	COL_2
2	A

4.9.2.2. GROUP(<input>, <groups>, <aggr>)

Computes aggregates over groups of rows. At least one group or one aggregate must exists. If no groups are present, then the aggregates are computed over all the rows.

Input:

- `input` - input table
- `groups` - optional list of zero-based grouping column indexes
- `aggr` - optional list of aggregates
  - `COUNT(ALL)` : count of rows in the group
  - `<aggr>(<col>)` : compute `<aggr>` for column `<col>` in each group, where the aggregate can be:
    - `COUNT` : number of not-null values
    - `MIN` : minimum value
    - `MAX` : maximum value
    - `SUM` : sum of numeric values
    - `AVG` : average of numeric values
  - `<aggr>(DISTINCT <col>)` : same as before, but only considering the distinct values

Output:

If there are not grouping columns, one single row with a column for each aggregate computed over all the input rows. Otherwise, one row for each group with as many columns as grouping columns, and the aggregate columns.

Examples:

```
GROUP(VALUES([INT,STRING],[[1,'A'],[2,'A'],[3,'B']]), [1],
      [COUNT(ALL), MAX(0), SUM(DISTINCT 0)])
```

COL_1	AGGR_1	AGGR_2	AGGR_3
A	2	2	3
B	1	3	3

4.9.2.3. SORT(<input>, <columns>)

Sorts the rows based on one or more sorting conditions.

Input:

- `input` - input table
- `columns` - list of sorting columns, with the zero-based index and the optional sort order (`ASC` or `DESC`)

Output:

The input rows sorted by the sorting criteria.

Examples:

```
SORT(VALUES([INT,STRING],[[1,'A'],[2,'A'],[3,'B']]), [1 DESC, 0])
```

COL_1	COL_2
3	B
1	A
2	A

4.9.2.4. EXTEND(<input>, <expressions>)

Creates new columns for each row by evaluating expressions over its current content.

Input:

- `input` - input table
- `expressions` - list of expressions to evaluate

Output:

For each input row, its current content plus the new columns with the result of the evaluation of the expressions.

Examples:

```
EXTEND(VALUES([INT],[[1],[2],[3]]), [%0 * 2, %COL_1 > 1])
```

COL_1	EXPR_1	EXPR_2
1	2	FALSE
2	4	TRUE
3	6	TRUE

4.9.2.5. DISTINCT(<input>)

Returns the distinct rows in a table.

Input:

- `input` - input table

Output:

The distinct rows from the input table.

Examples:

```
DISTINCT(VALUES([INT,STRING],[[1,'A'],[3,'B'],[1,'A']]))
```

COL_1	COL_2
1	A
3	B

4.9.2.6. UNION(<input$_1$>, ..., <input$_n$>)

Concatenates two or more compatible tables. Two tables are compatible if they have the same number of columns with matching data types for each column.

Input:

- `input`$_i$ - an input table

Output:

All the rows of all the input tables

Examples:

```
UNION(VALUES([INT],[[1],[3]]),VALUES([INT],[[2]]))
```

COL_1
1
3
2

4.9.2.7. PRODUCT(<left>, <right>)

Cartesian product of two tables.

Input:

- `left` - left input table
- `right` - right input table

Output:

The cartesian product between the rows of both inputs. Each output row has the columns of its left component plus the columns of its right component.

Examples:

```
PRODUCT(VALUES([INT,STRING],[[1,'A'],[2,'B']]),VALUES([INT],[[3],[4]]))
```

LEFT_1	LEFT_2	RIGHT_1

1	A	3
1	A	4
2	B	3
2	B	4

4.9.2.8. JOIN(<left>, <right>, <condition>, <options>)

A join is a selection over a cartesian product of two tables.

Input:

- `left` - left input table
- `right` - right input table
- `condition` - boolean join expression, the supported operands are:
  - `=, <>, <, <=, >, >=` : comparison operators between two zero-based column indexes (prefixed with a % character).
  - `X AND Y` : logical AND between X and Y
  - `X OR Y` : logical OR between X and Y
  - `NOT X` : negates X
  - `(X)` : evaluates X, it can be used to change the evaluation priorities
- `options` - optional property map (for extensions such as semi-joins, join algorithm, etc.)

Output:

A selection of the rows of the cartesian product between the rows of both inputs. Each output row has the columns of its left component plus the columns of its right component.

Examples:

```
JOIN(VALUES([STRING,INT],[['A',1],['B',2]]),
     VALUES([INT],[[3],[2]]),%1<%2,NULL)
```

LEFT_1	*LEFT_2*	*RIGHT_1*
A	1	3
A	1	2
B	2	3

4.10. Parsing CloudMdsQL query plans to CQA query plans

This section shows the procedure of converting from a CloudMdsQL query to a CQA query. For example, the following query accesses two data stores, one relational and another native (MongoDB). The result of both subqueries is then joined in a single result.

```
T1( a int, b string )@DB1 = ( SELECT a, b FROM tbl WHERE id > 100 )
T2( a int, c string )@DB2 = {* db.find( { id: 200 } ) *}
SELECT T1.a, T1.b, T2.c
FROM T1 JOIN T2 ON T1.a = T2.a
```

The query plan represented in JSON is:

```
{
   "sub": [
      {
         "name": "t1",
         "plan": {
            "op": "TRANSFORM",
            "plan": {
               "op": "PROJECT",
               "operands": [
                  {
                     "op": "SELECT",
                     "operands": [
                        { "op": "TABLEREF", "name": "tbl" }
                     ],
                     "filter": {
                        "expr": "func",
                        "function": ">",
                        "operands": [
                           { "expr": "colref", "colref": [ "id" ] },
                           { "expr": "const", "datatype": "INT", "value": "100" }
                        ]
                     }
                  }
               ],
               "columns": [
                  {
                     "value": { "expr": "colref", "colref": [ "a" ] },
                     "name": "a"
                  },
                  {
                     "value": { "expr": "colref", "colref": [ "b" ] },
                     "name": "b"
                  }
               ]
            },
            "datastore": "db1",
            "signature": [ "INT", "STRING" ]
         }
      },
      {
         "name": "t2",
         "plan": {
            "op": "NATIVE",
            "datastore": "db2",
            "code": " db.find( { id: 200 } ) ",
            "signature": [ "INT", "STRING" ]
         }
      }
   ],
   "plan": {
      "op": "PROJECT",
      "operands": [
         {
            "op": "JOIN",
            "operands": [
               {
                  "op": "PROJECT",
                  "operands": [
                     {
                        "op": "CALL",
                        "sub": "t1"
```

```
                  }
               ],
               "columns": [
                  {
                     "value": { "expr": "colref", "colref": [ "0" ] }
                  },
                  {
                     "value": { "expr": "colref", "colref": [ "1" ] }
                  }
               ]
            },
            {
               "op": "PROJECT",
               "operands": [
                  {
                     "op": "CALL",
                     "sub": "t2"
                  }
               ],
               "columns": [
                  {
                     "value": { "expr": "colref", "colref": [ "0" ] }
                  },
                  {
                     "value": { "expr": "colref", "colref": [ "1" ] }
                  }
               ]
            }
         ],
         "type": "INNER",
         "condition": {
            "expr": "func",
            "function": "=",
            "operands": [
               { "expr": "colref", "colref": [ "0", "0" ] },
               { "expr": "colref", "colref": [ "0", "1" ] }
            ]
         },
         "result": [
            { "expr": "colref", "colref": [ "0", "0" ] },
            { "expr": "colref", "colref": [ "1", "0" ] },
            { "expr": "colref", "colref": [ "0", "1" ] },
            { "expr": "colref", "colref": [ "1", "1" ] }
         ]
      }
   ],
   "columns": [
      {
         "value": { "expr": "colref", "colref": [ "0" ] },
         "name": "a"
      },
      {
         "value": { "expr": "colref", "colref": [ "1" ] },
         "name": "b"
      },
      {
         "value": { "expr": "colref", "colref": [ "3" ] },
         "name": "c"
      }
   ]
  }
}
```

The procedure to convert the query plan is:

1. Identify all the dependencies from data stores:
    a. hierarchical dependencies (from the query tree)

   b. from `REFERENCING` clauses
2. From the data store dependencies, identify which ones will be cached into the Table Store, and those that will flow in the pipeline without caching
3. Generate the CQA query. In particular:
   a. each native and non native query is a `STREAM` operator
   b. each Python query is a `STREAM` operator
   c. cached data stores are accessed through the `STORE` functions

Thus, the sample query is translated into:

```
LET @t1 = STREAM('native',
                 [INT,STRING],
                 ['db1','
                  "plan": {
                      "op": "TRANSFORM",
                      "plan": {
                        "op": "PROJECT",
                        "operands": [
                            {
                              "op": "SELECT",
                              "operands": [
                                  { "op": "TABLEREF", "name": "tbl" }
                              ],
                              "filter": {
                                  "expr": "func",
                                  "function": ">",
                                  "operands": [
                                    {"expr": "colref", "colref": [ "id" ] },
                                    {"expr":"const","datatype":"INT","value":"100"}
                                  ]
                              }
                            }
                        ],
                        "columns": [
                            {
                              "value": { "expr": "colref", "colref": [ "a" ] },
                              "name": "a"
                            },
                            {
                              "value": { "expr": "colref", "colref": [ "b" ] },
                              "name": "b"
                            }
                        ]
                      },
                      "datastore": "db1",
                      "signature": [ "INT", "STRING" ]
                  }'],
                 NULL),
    @t2 = STREAM('native',
                 [INT,STRING],
                 ['db2','
                  "plan": {
                    "op": "NATIVE",
                    "datastore": "db2",
                    "code": " db.find( { id: 200 } ) ",
                    "signature": [ "INT", "STRING" ]
                  }'],
                  NULL)
    IN RENAME(PROJECT(
                   JOIN(@t1, @t2,%0=%0, NULL),
                   [0,1,3]),
              ['a','b','c'])
```

5. Wrappers: Query Engine interaction with the Data Stores

A wrapper provides the interface that attaches a data store to the CloudMDS Query Engine (CQE). It handles fragments of the query plan that are intended for execution in a data store and delivers interim results in the appropriate format.

Considering the diversity of data stores, their functionality is presented to the CQE using a JSON schema, which describes the query plan fragments that can be handled. These range from a simple native query string, opaque to the CQE, to supporting relational operations such as join and grouped aggregation, which are directly processed by the Operator Engine. The schema is a sub-set of the internal Query Plan schema described in section Appendix A.

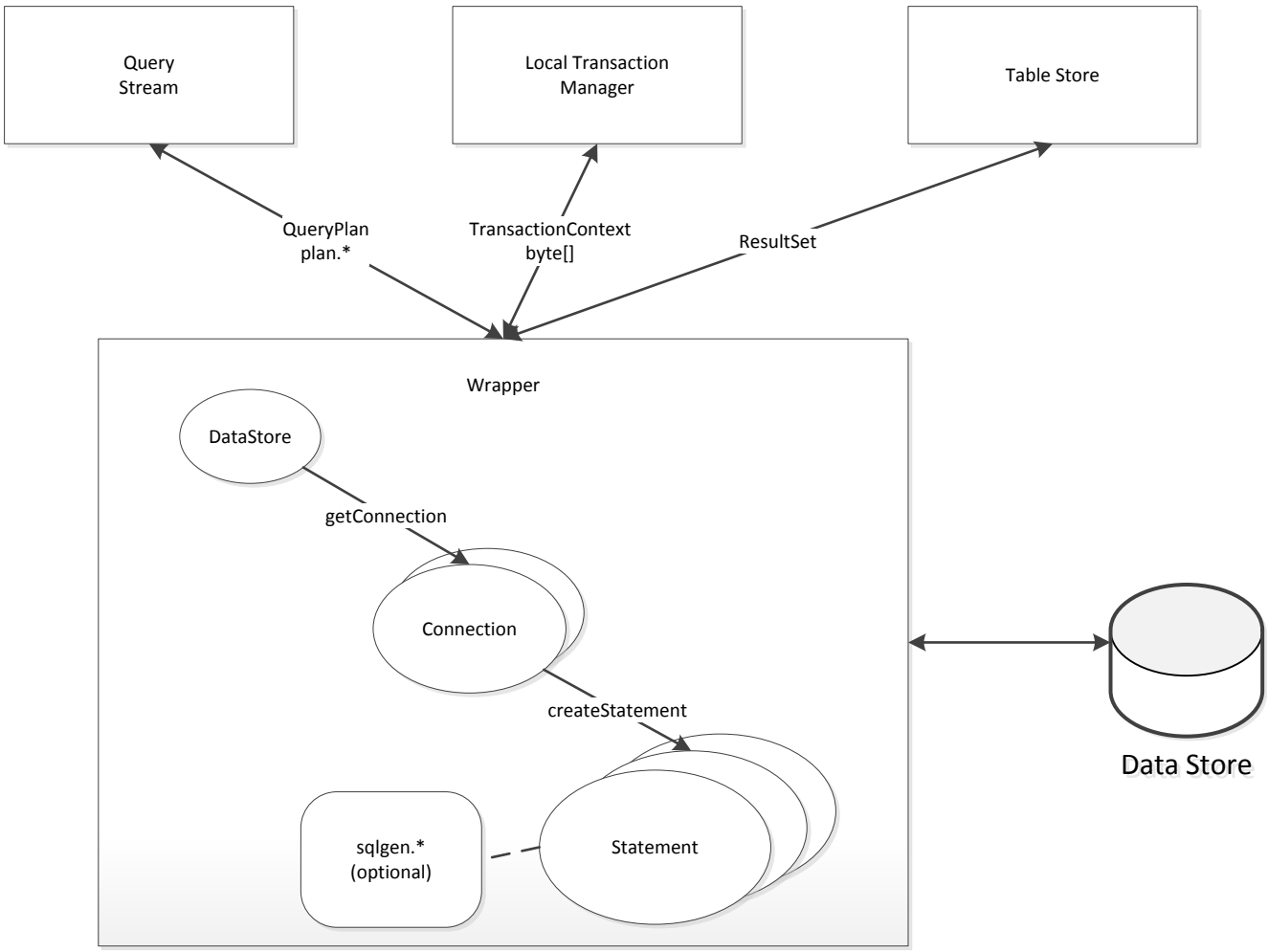
5.1. Rationale

The wrapper API and implementations are based on the Java platform and conceptually similar to the standard JDBC interface, namely, regarding the usage of Driver/Connection/Statement objects to provide nested context for query execution, and a ResultSet to iterate over result data.

This similarity helps wrapper developers as JDBC is expected to be the interface to many data stores or, at least, familiar to most wrapper developers. The wrapper interface differs from JDBC in three important aspects. First, in contrast to JDBC that is supposed to be useful to a wide range of client programs and thus offers different ways to accomplish the same goal, the wrapper API is much simpler. Second, as the only client to this API is the CQE, the wrapper focuses on performance, such as facilitating batch processing by operators. Finally, it has to provide access to named tables in the Table Store and to the Transactional Context.

5.2. Wrapper architecture and interfaces

The internal architecture of a wrapper and the interactions with the main CQE components are depicted in the following figure:



The main entry point to the wrapper is the DataStore interface, which provides Connection contexts for the execution of Statements. The query is received as a Java mapping of the JSON data model originating from the Query Stream. The Local Transaction Manager is exposed as a current Transaction Context and is then given write sets as opaque byte arrays. Interaction with the Table Store is, in both directions, through the ResultSet batch iterator interface.

Classes and interfaces that compose the wrapper interface are grouped in four different packages briefly described in this section. A complete specification is available in Javadoc format with the source code.

5.2.1. eu.coherentpaas.cqe.datastore

This package contains the interfaces that must be implemented by each wrapper, namely:

- DataStore: This represents an instance of a data store within the CQE. It is the main entry point into the wrapper.
- Connection: This represents a single-threaded entry point to the data store. Therefore, different Connection instances are used for concurrent usage of the same data store.
- Statement: This holds cached state for a query, that can be executed multiple times.

- DataStoreMetadata: Allows the CQE to query the data store for meta-data, for instance, the schema of returned data.

5.2.2. eu.coherentpaas.cqe

This package exposes CQE interfaces to the wrapper for services provided by or data exchanged with the CQE, namely:

- ResultSet: Provides a batch iterator to retrieve data, in both directions, between the data store wrapper and the CQE.
- TransactionalContext: Encapsulates the current holistic transactional context that is propagated from the CQE to the data stores.
- NativeQueryPlan/TransformQueryPlan: Root classes for native and transformable query plans being sent to the wrapper.

5.2.3. eu.coherentpaas.cqe.plan

This package contains a mapping to Java of the query plan JSON schema using the Object Mapping feature of the Jackson JSON framework for Java. It is thus used to provide the wrappers with a convenient interface to receive query plans and, in particular, to transform them using the Visitor pattern.

5.2.4. eu.coherentpaas.cqe.sqlgen

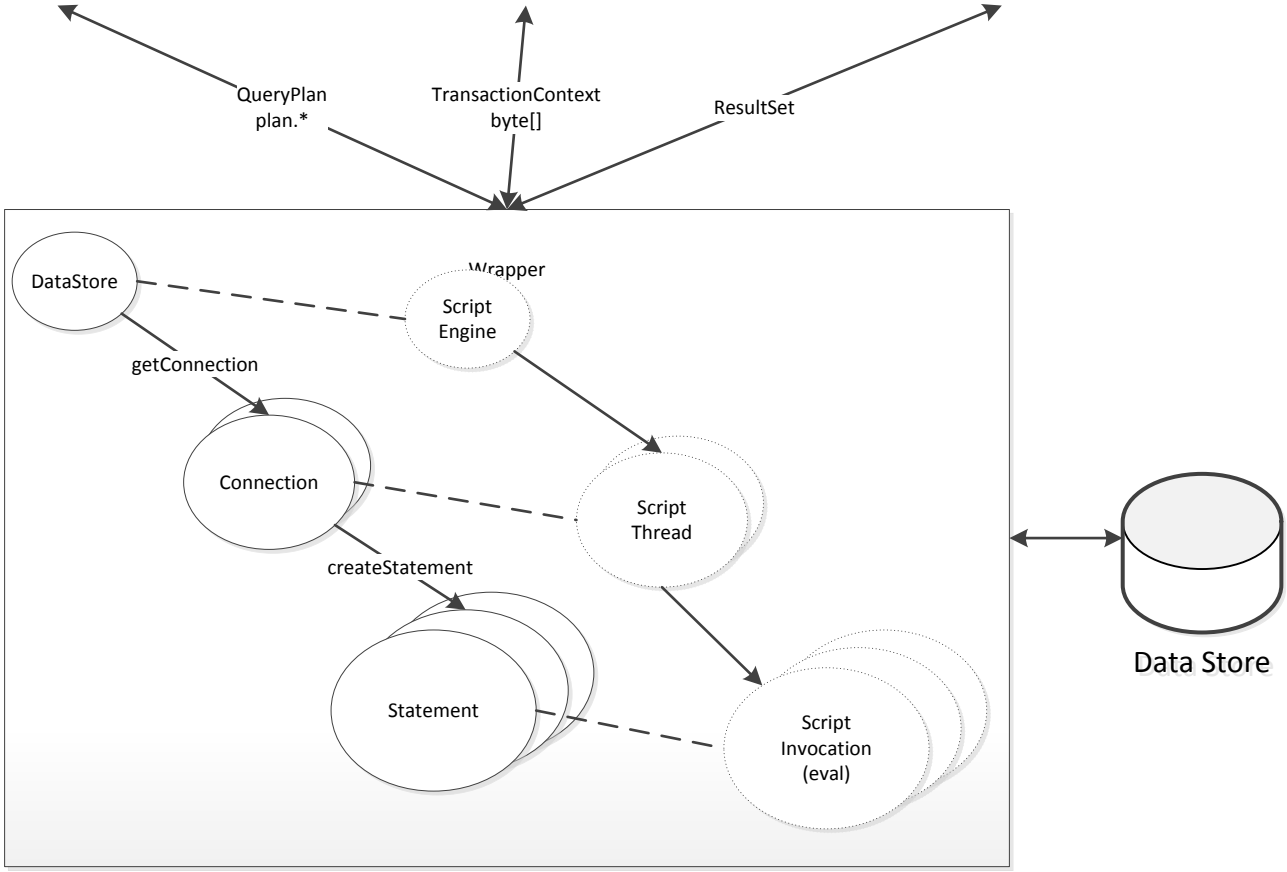
This package contains a generic converter from transformable query plan fragments to SQL. It is thus useful only, even if not mandatory, for data stores that handle a SQL dialect. It is based on the eu.coherentpaas.cqe.plan package and the Visitor pattern.

5.3. Common wrapper implementations

By providing the wrapper a query plan fragment that respects a sub-set of the query plan JSON schema, the wrapper has a range of options for receiving queries and having them executed, as explained below. A wrapper can thus provide any of these alternatives, or even more than one of them simultaneously.

5.3.1. Native query, client-side execution

The wrapper specifies through JSON schema that it handles native queries (i.e., a JSON document that has a NATIVE node as root). The wrapper then uses an embedded script engine to handle these queries, exposing in the script language primitives for returning the results in the CQE data model. The query has thus a large amount of control on how the mapping is performed and makes this the preferred model for data stores with a significantly different data model that cannot be automatically mapped to relational data.
This is depicted in the following Figure:



5.3.2. Native query, server-side execution

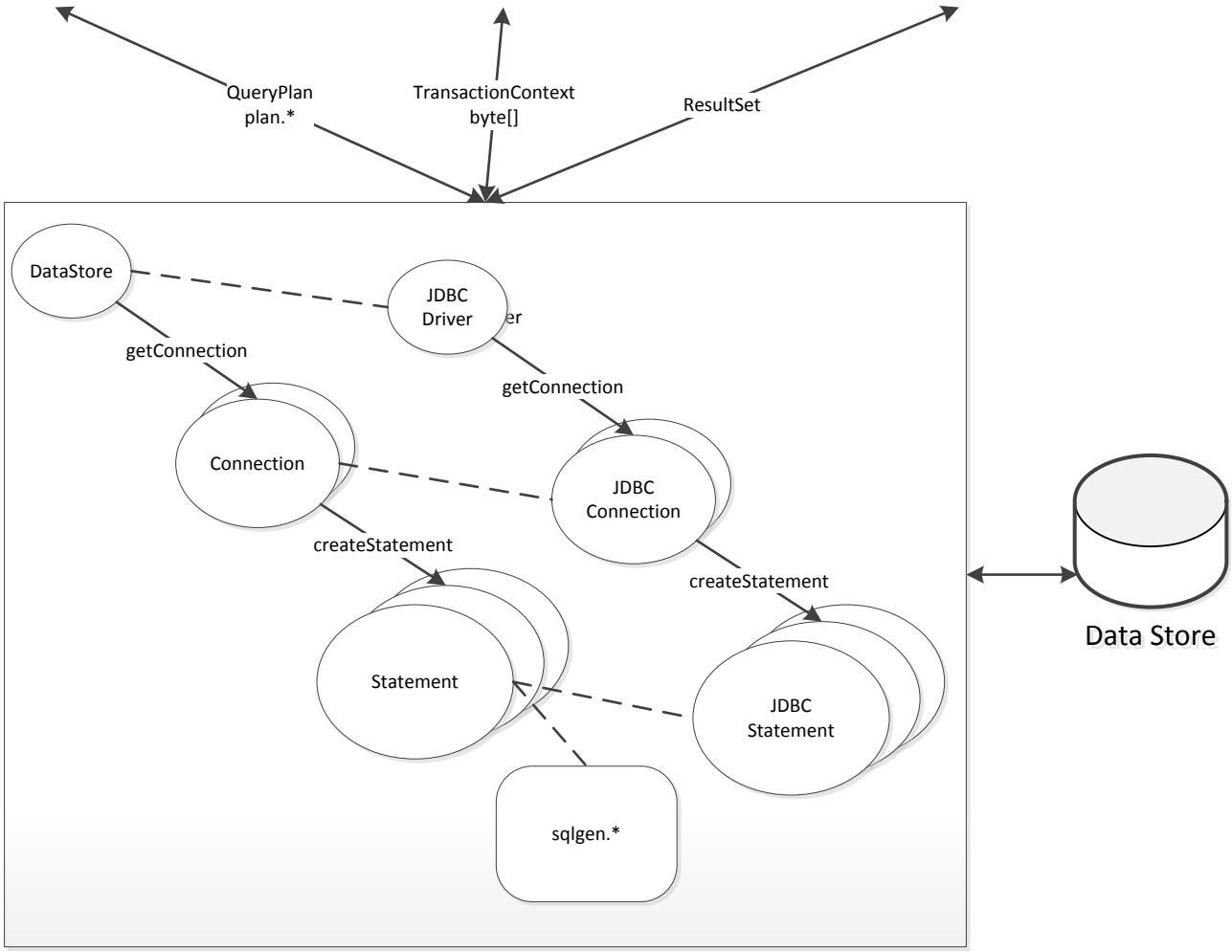
The wrapper specifies through JSON schema that it handles native queries (i.e., a JSON document that has a NATIVE node as root). These queries are shipped to the data store for execution. The wrapper's main task is to map results to the CQE data model, which has to be done after the query has run and without its intervention. This approach is appropriate, for instance, for non-standard SQL statements to be passed to a SQL data store.

5.3.3. Transform query, custom interface

In this case, the wrapper specifies through JSON schema that it handles a sub-set of transform query plans, such as table scans with filters (i.e., a JSON document that has a TRANSFORM node as root). These plans are then interpreted in the wrapper, invoking data store interfaces. This approach is useful, for instance, for key-value stores with very simple data models and APIs that can do server-side filtering.

5.3.4. Transform query, SQL interface

The wrapper specifies through JSON schema that it handles transform query plans (i.e., a JSON document that has a TRANSFORM node as root). These plans are then converted to SQL statements using the eu.coherentpaas.cqe.sqlgen utility package sent for server-side execution in the data store. This approach is expected to be the preferred for SQL-like data stores and is depicted in the following Figure:



5.4. Operation

5.4.1. Startup and shutdown

A wrapper is configured in the CQE by providing the name of a class implementing eu.coherentpaas.cqe.datastore.DataStore, all required class path elements, and a properties file. The content of this file is not in any way examined by the CQE and the meaning of the property names and values is exclusively for use by the wrapper.
The wrapper is started by the CQE, by creating an instance of this class and invoking the *start* method with the corresponding properties file. This instance is thus associated with an instance of the data store, as specified in a wrapper specific format in the properties file. It will remain valid until the CQE invokes the *close* method on the same instance, which should release all resources used by the wrapper.

5.4.2. Connections and statements

A data store instance is used through connections. Each connection is obtained with a *getConnection* method, used by a single thread, and returned to the wrapper by invoking the *close* method in the connection object. The CQE should assume that obtaining and releasing connection objects is a lightweight operation and is not expected to perform pooling. It is up to the wrapper implementation to ensure this, possibly by keeping a

connection pool itself, if this is considered necessary. For instance, for SQL-like databases this shouldn't be necessary if the underlying JDBC driver itself does pooling.
Connections can be used to prepare and execute one or more statements. Only one statement can be in execution at any given moment. Executing a second statement will cancel pending result sets, which will not return further data. Execution of a statement thus allows the following steps:

- Optionally, invocations of *useNamedTable* method inform the wrapper that the next query should be allowed to use a named table, referred in an implementation specific manner by that name. This method provides a callback to be used later during execution to retrieve data from the Table Store. The callback is not valid when not currently invoking the execute method.
- A mandatory invocation of *prepare* provides the query plan, that can be either a native or a transform fragment, according to what is supported by the wrapper. It optionally includes types and names of named parameters.
- Optionally, invocations of *set** methods provide values for named parameters referred to by the query plan. These are valid only for the next execution.
- The invocation of *execute* starts execution, providing a transactional context to be relayed to the data store. It returns a result set to retrieve results, if any expected.
- Results are retrieved with *next* methods. When no more data is required, the *close* method is invoked to release resources used for that query.
- Finally, an invocation to *getWriteSet* obtains an opaque representation of the write set resulting from the execution, if any, that is relayed back to the transaction manager.

It is expected that as much work as possible is done during the invocation of *prepare*, being reused for multiple invocations of execute. For instance, a SQL-like data store wrapper will convert the query plan to a SQL query and send it to the server for compilation, using a JDBC prepared statement.

5.4.3. Named tables and parameters

Parameters are described by the query plan and their values are supplied before execution. Regardless of the wrapper implementation, such as using an embedded client-side script engine or a server-side query language, there is some implementation dependent way to bind them to their values. For instance, in SQL this can be done with prepared statements.
The same can be done for named tables when the query is executed in the client side. It requires only that the named table is exposed in a way that calls back into the wrapper to retrieve data, which is obtained from the Table Store through the Parameterized interface.
This may be harder to accomplish for server-side execution, when the interface does not support callback. This is the case for JDBC and thus for SQL-like data stores. In this case, the wrapper can still do the following: When *execute* method is invoked and a named table is being used, the wrapper can download all data from the Table Store and upload it to a temporary table. After that, the execution proceeds with the appropriate data. Finally, after execution completes, temporary tables are dropped.

6. References

[Boncz05] – P. Boncz, M. Zukowski, and N. Niels. *MonetDB/X100: Hyper-Pipelining Query Execution*. CIDR. Vol. 5. 2005.

[Graefe94] - G. Graefe. *Volcano - an extensible and parallel query evaluation system.* IEEE TKDS, 6(1):120–135, 1994.

[Tomasic98] - A. Tomasic, L. Raschid, P. Valduriez. *Scaling Access to Heterogeneous Data Sources with DISCO*. IEEE Transactions on Knowledge and Data Engineering, 10(5): 808-823, 1998.

Appendix A. Query Plan JSON Schema

This section contains the JSON schema as per draft 4 of http://json-schema.org, which describes the format of the query execution plan, produced by CloudMdsQL compiler and consumed by the execution engine.

```
{
    "description": "Schema for the CoherentPaaS query execution plan",
    "type": "object",
    "required": [ "plan" ],
    "properties": {
        "name": { "type": "string" },
        "text": { "type": "string" },
        "sub":  {
            "type": "array",
            "items": {
                "type": "object",
                "required": [ "name", "plan" ],
                "properties": {
                    "name": { "type": "string"},
                    "params": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "required": [ "name", "datatype" ],
                            "properties": {
                                "name": { "type": "string" },
                                "datatype": { "$ref": "#/definitions/datatype" }
                            }
                        }
                    },
                    "plan": { "$ref": "#/definitions/operation" }
                }
            }
        },
        "plan": { "$ref": "#/definitions/operation" }
    },
    "definitions": {
        "operation": {
            "type": "object",
            "oneOf": [
                { "$ref": "#/definitions/call" },
                { "$ref": "#/definitions/project" },
                { "$ref": "#/definitions/select" },
                { "$ref": "#/definitions/aggregate" },
                { "$ref": "#/definitions/join" },
                { "$ref": "#/definitions/python" },
                { "$ref": "#/definitions/native" },
                { "$ref": "#/definitions/transform" },
                { "$ref": "#/definitions/tableref" }
            ]
        },
        "expression": {
            "type": "object",
            "oneOf": [
                { "$ref": "#/definitions/cref" },
                { "$ref": "#/definitions/const" },
                { "$ref": "#/definitions/param" },
                { "$ref": "#/definitions/func" }
            ]
        },
        "call": {
            "type": "object",
            "required": ["op", "sub"],
```

```
        "properties": {
            "op": { "type": "string", "pattern": "CALL" },
            "sub": { "type": "string" },
            "params": {
                "type": "array",
                "items": {
                    "type": "object",
                    "required": ["name", "value"],
                    "properties": {
                        "name": { "type": "string" },
                        "value": { "oneOf": [
                            { "$ref": "#/definitions/const" },
                            { "$ref": "#/definitions/param" }
                        ] }
                    }
                }
            }
        }
    },
    "project": {
        "type": "object",
        "required": ["op", "operands", "columns"],
        "properties": {
            "op": { "type": "string", "pattern": "PROJECT" },
            "operands": { "type": "array",
                          "items": { "$ref": "#/definitions/operation" },
                          "minItems": 1, "maxItems": 1 },
            "columns": {
                "type": "array",
                "items": {
                    "type": "object",
                    "required": ["name", "value"],
                    "properties": {
                        "name": { "type": "string" },
                        "value": { "$ref": "#/definitions/expression" }
                    }
                },
                "minItems": 1
            }
        }
    },
    "select": {
        "type": "object",
        "required": ["op", "operands", "filter"],
        "properties": {
            "op": { "type": "string", "pattern": "SELECT" },
            "operands": { "type": "array",
                          "items": { "$ref": "#/definitions/operation" },
                          "minItems": 1, "maxItems": 1 },
            "filter": { "$ref": "#/definitions/expression" }
        }
    },
    "aggregate": {
        "type": "object",
        "required": ["op", "operands"],
        "properties": {
            "op": { "type": "string", "pattern": "AGGR" },
            "operands": { "type": "array",
                          "items": { "$ref": "#/definitions/operation" },
                          "minItems": 1, "maxItems": 1 },
            "groupby": { "type": "array", "items": { "type": "integer" } },
            "aggregates": {
                "type": "array",
                "items": {
                    "type": "object",
                    "required": ["aggregate"],
                    "properties": {
                        "aggregate": { "enum": [ "COUNT_ALL", "COUNT", "SUM",
                                                 "AVG", "MIN", "MAX" ] },
                        "colref": { "type": "integer" }
```

```
                }
            }
        }
    }
},
"join": {
    "type": "object",
    "required": ["op", "operands", "type"],
    "properties": {
        "op": { "type": "string", "pattern": "JOIN" },
        "operands": { "type": "array",
                      "items": { "$ref": "#/definitions/operation" },
                      "minItems": 2, "maxItems": 2 },
        "type": { "enum": [ "INNER", "LEFT", "RIGHT", "FULL" ] },
        "condition": { "$ref": "#/definitions/expression" },
        "result": { "type": "array",
                    "items": { "$ref": "#/definitions/cref" },
                    "minItems": 1 }
    }
},
"python": {
    "type": "object",
    "required": ["op", "code", "signature"],
    "properties": {
        "op": { "type": "string", "pattern": "PYTHON" },
        "code": { "type": "string" },
        "signature": { "type": "array",
                       "items": { "$ref": "#/definitions/datatype" },
                       "minItems": 1 },
        "references": { "type": "array", "items": { "type": "string" } }
    }
},
"native": {
    "type": "object",
    "required": ["op", "datastore", "code"],
    "properties": {
        "op": { "type": "string", "pattern": "NATIVE" },
        "datastore": { "type": "string" },
        "code": { "type": "string" },
        "signature": { "type": "array",
                       "items": { "$ref": "#/definitions/datatype" } },
        "references": { "type": "array",
                        "items": { "type": "string" } }
    }
},
"transform": {
    "type": "object",
    "required": ["op", "datastore", "plan", "signature"],
    "properties": {
        "op": { "type": "string", "pattern": "TRANSFORM" },
        "datastore": { "type": "string" },
        "plan": { "$ref": "#/definitions/operation" },
        "signature": { "type": "array",
                       "items": { "$ref": "#/definitions/datatype" },
                       "minItems": 1 }
    }
},
"tableref": {
    "type": "object",
    "required": ["op", "name"],
    "properties": {
        "op": { "type": "string", "pattern": "TABLEREF" },
        "name": { "type": "string" }
    }
},
"cref": {
    "type": "object",
    "required": ["expr", "colref"],
    "properties": {
        "expr": { "type": "string", "pattern": "colref" },
```

```
                "colref": { "type": "array",
                            "items": { "type": "string" },
                            "minItems": 1 }
            }
        },
        "const": {
            "type": "object",
            "required": ["expr", "datatype", "value"],
            "properties": {
                "expr": { "type": "string", "pattern": "const" },
                "datatype": { "$ref": "#/definitions/datatype" },
                "value": { "type": "string" }
            }
        },
        "param": {
            "type": "object",
            "required": ["expr", "name"],
            "properties": {
                "expr": { "type": "string", "pattern": "param" },
                "name": { "type": "string" }
            }
        },
        "func": {
            "type": "object",
            "required": ["expr", "function"],
            "properties": {
                "expr": { "type": "string", "pattern": "func" },
                "function": { "type": "string" },
                "operands": { "type": "array",
                              "items": { "$ref": "#/definitions/expression" } }
            }
        },
        "datatype": { "enum": [ "STRING", "INT" ] }
    }
}
```