Optique™

Project N°:	**FP7-318338**
Project Acronym:	**Optique**
Project Title:	**Scalable End-user Access to Big Data**
Instrument:	**Integrated Project**
Scheme:	**Information & Communication Technologies**

Deliverable D7.2
Techniques for Distributed Query Planning and Execution: Continuous/Streaming and Temporal Queries

Due date of deliverable:	(T0+24)
Actual submission date:	November 2, 2014

Start date of the project: **1st November 2012**	Duration: **48 months**
Lead contractor for this deliverable:	**UoA**
Dissemination level:	**PU** – Public

Final version

Executive Summary:

Techniques for Distributed Query Planning and Execution: Continuous/Streaming and Temporal Queries

This document summarises deliverable D7.2 of project FP7-318338 (Optique), an Integrated Project supported by the 7th Framework Programme of the EC. Full information on this project, including the contents of this deliverable, is available online at `http://www.optique-project.eu/`.

List of Authors

Konstantina Bereta (UoA)
Dimitris Bilidas (UoA)
Herald Kllapi (UoA)
Christos Mallios (UoA)
Alexandros Papadopoulos (UoA)
Christoforos Svingos (UoA)
Dimitris Theodosakis (UoA)
Manolis Koubarakis (UoA)
Yannis Ioannidis (UoA)

List of Contributors

Ralf Möller (UoL)
Guohui Xiao (FUB)

Contents

Chapter 1

Introduction

This deliverable reports the progress made in work package WP7 during the second year of the Optique project. According to the initial plan, during the second year, the main effort should be put in task 7.2, entitled "Query planning and execution techniques for continuous and temporal OBDA queries". Indeed, a large portion of the work described in this deliverable is about the stream processing capabilities that have been added to the distributed query execution component. But at the same time, as it has become evident during the course of the project and especially after the first review, during the second year we should continue the work regarding optimization of the query processing and also, as a need that emerged from the use cases of the project, we should tackle the requirement for federated query execution. In Chapter 2 we describe the progress in the optimization process and the federated execution, whereas in Chapter 3 we describe the progress in continuous and temporal queries.

More specifically, in Chapter 2, we present improvements and results regarding general optimizations of the system, but also optimizations with regard to specific aspects, like the federated or analytical queries. General optimizations include the pipelined execution of operators in Section 4.2, the cost model in Section 2.1 and the multi-query optimization presented in the last paragraph of Section 2.2. Regarding federated execution, in Section 2.2.2 we present the query decomposition with respect to this case and in section 2.3 we present the component of the system that provides caching over the network in order to achieve efficient execution of federated queries. In Section 2.5 we present efficient techniques for tree-like execution of group-by operators contained in analytical queries. In Chapter 3, we present the streaming functionality of the engine, the streaming operators we support, and the language. Furthermore, we show how to use the engine with remote sources and present some experiments that show the efficiency. In Chapter 4, we present some large-scale experiments and compare with state-of-the-art systems. Finally, we show the efficiency of the JIT techniques we use in the engine.

Chapter 2

Improvements and Optimization for Federated and Analytical Queries

In this chapter, we present several optimizations and improvements we did to the system. In Section 2.1 we present the component that is responsible for storing and updating the statistics for the tables. We describe the cost model that it employs and the distributed analyzer which combines the partial statistics from the table partitions and combines them to generate an overall estimation of the whole table. We then proceed to describe the decomposition of input queries into the dataflow language in Section 2.2. The component responsible for the decomposition uses the component which keeps the statistics and, depending on the mode of operation, produces a sequence of queries in the dataflow language of ADP. In this section, we also present the initial results regarding multi query optimization in the system, as this aspect of optimization is integrated into the decomposition process. In Section 2.3 we present the cashing over the network method that we use in order to achieve efficient execution of federated queries. We close this chapter with a method for the efficient execution of queries using tree execution plans that contains aggregations, distributive, and algebraic functions. Using this technique, we can achieve near-interactive response times for very large datasets, like the ones we have in the Statoil usecase.

2.1 Cost Model and Table Statistics

In this section, we present the optimization techniques we use to translate the high-level SQL query into the dataflow language of the system. We describe the component of the system which is responsible for the estimation of the cost of the operators during execution. We present the statistics that we store for each table in the database and the main characteristics of the cost model, along with the distributed analyzer, which combines statistics from different nodes of the system. We also present some statistics from the SLEGGE database. Finally, we present the grouping of star joins, which aims to increase parallelism of execution in a distributed environment.

2.1.1 Query Optimization

Query Optimizer Architecture

The query optimizer of the system [19] has two main parts:

- **Plan generator**: It produces plans from the initial plan which is constructed based on the join predicates that exist in the given high-level sql query. Each plan is an ordered set of star join patterns. Each join pattern is defined by the star's central table along with the joining tables. One star may have multiple configurations, one for each central table.
- **Estimator**: It is based on histograms for selectivity estimation and on a cost model tailored to the ADP system to compute costs when a table needs to be repartitioned or replicated. For the time being,

a table can lie in one or in N partitions (where N is the number of the existing virtual machines). We are currently working on a more sophisticated approach to better leverage parallelism. Moreover, the ADP system maintains a metadata store where statistics and other useful information are being gathered to aid the query optimization process.

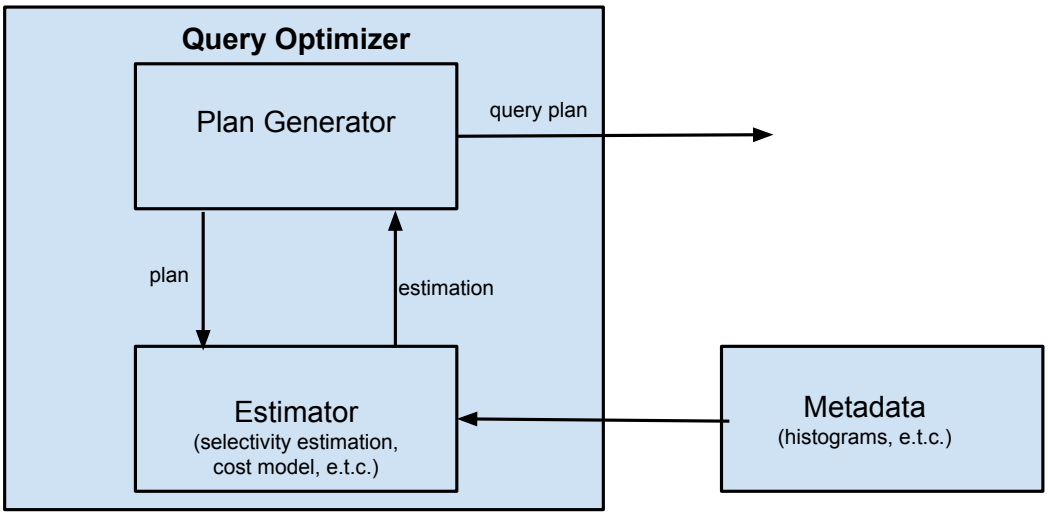

Figure 2.1: The architecture of the optimizer

Selectivity Estimation

When the optimizer estimates the cost of a given SQL query, it searches for statistics in the metadata store. At this time, we use the heuristic that filters are executed first in order to reduce the intermediate results and so as to aid the execution of the following join predicates.

A histogram [18] is an ordered set of buckets, each of which contains information about the distinct number of values in the bucket and the frequency, meaning the mean number of records in this bucket. As far as the histogram construction policy is concerned, we choose to use equi-depth histograms. Inside each bucket the uniform distribution is assumed. From the estimator's perspective, a table is a set of histograms, one for each column. Other statistics taken into consideration are the number of records for each table and the length of each column in bytes.

In every step, the estimator depicts intermediate results to histograms. For example, in case of a filter operator in a specific column, it applies the filter to the corresponding histogram to produce an estimation of the current state of the column. In the same way, in case of a join operator, it joins the histograms of the joining columns and produces a histogram that depicts the data distribution after applying the operator. In this way, the query optimizer holds a snapshot of the current database state.

Cost Model

In order to estimate the cost of a given query plan, the estimator uses a cost model. The main objective of the query optimizer is to find a good order of the star join patterns. So, for each join pattern it must determine both the central table and the degree of parallelism, meaning the number of partitions that the hash join will be done.

Each logical table that participates in a star join pattern needs to have the same number of physical partitions. To search in this space, the cost of replication or repartition needs to be estimated because network cost and data shipping are usually the dominant factor in a distributed environment. In the end of the optimization process, a plan would be an order of star join patterns, each of them partitioned on the most suitable number of partitions and with the most suitable table as a center based on the estimation that takes into account not only the local cost in each node (cpu time and disk I/O) but only the network cost.

2.1.2 Distributed Statistics

In order to gather statistics to aid the query optimizer, we have implemented a distributed analyze functionality. In the ADP system, for each a specific table and column, every node that contains a table's partition, is capable of building histograms and then, all the relating histograms are unioned in the master node, which is responsible for the statistics management. In order to build a histogram, a node samples data lying in the specified table partition and column.

2.1.3 SLEGGE Database Statistics

We extracted some statistics from the SLEGGE schema and we constructed the following chart:

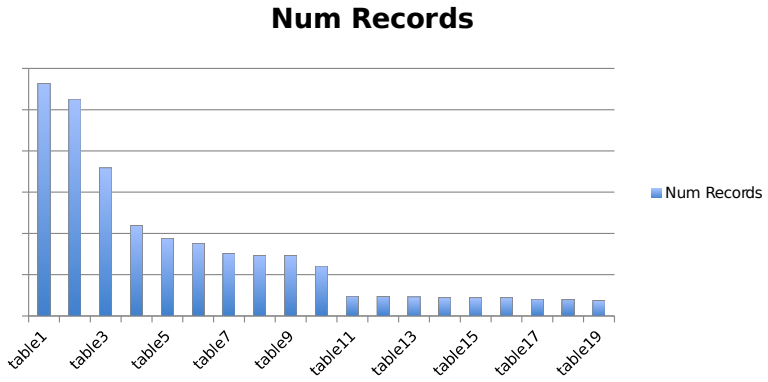

Figure 2.2: SLEGGE tables (actual numbers are hidden)

This histogram shows that only a small fraction of the 1684 total tables contains a significant number of records. Moreover, we observed that most of the existing tables (1150) are completely empty (0 records) and also there are many tables ($\tilde{4}20$) containing only a small number of records (1-10000). It is then quite obvious that such information is very important. For example, in the initial data placement, the largest tables can be partitioned into N partitions (where N is the number of the available virtual machines) and the smallest ones can be replicated in every node. In this way, joins are executed very efficiently as we show in the next sections.

2.1.4 Star Join Optimization

Given an SQL query which needs to be executed in the ADP system, the optimizer discovers all possible star join patterns[14]. As a star join pattern we define a subgraph of the join graph in which all the participant tables are partitioned (using the same hash function) on the same column and have the same number of partitions. Note that all the join predicates must be on the same partitioned column.

In a distributed environment, executing a star join subgraph instead of each join separately, is very efficient. Imagine a query that involves a star join with N tables that satisfies the above properties (all tables partitioned on the same attribute and have the same number of partitions). In such a case, we need to produce only one distributed query instead of producing N distinct distributed queries.

As an example, assume the following schema:

```
table1(id1, fld1) partitioned on id1 to 4 partitions
table2(id2, fld2) partitioned on id2 to 4 partitions
table3(id3, fld3) partitioned on id3 to 4 partitions
table4(id4, fld4) partitioned on id4 to 4 partitions
```

and this high level SQL query:

```
select  t1.fld1, t2.fld2
from table1 t1, table2 t2, table3 t3, table4 t4
where t1.id1 = t2.id2 and t2.id2 = t3.id3 and t3.id3 = t4.id4
```

A possible resulting ADP distributed query could be:

```
distributed create temporary table result to 4 as direct
select t1.fld1, t2.fld2
from table1 t1, table2 t2, table3 t3, table4 t4
where t1.id1 = t2.id2 and t1.id1 = t3.id3 and t1.id1 = t4.id4
```

The following figures depict all possible star join plans that can be reulted:

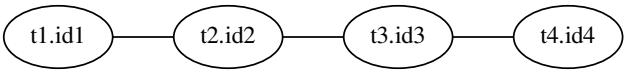

Figure 2.3: Initial join graph

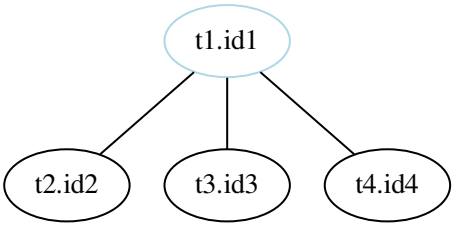

Figure 2.4: Join star pattern with table1 as the center

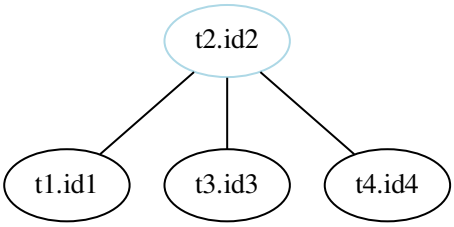

Figure 2.5: Join star pattern with table2 as the center

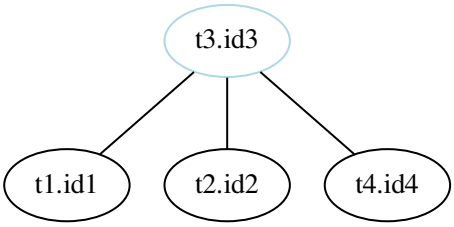

Figure 2.6: Join star pattern with table3 as the center

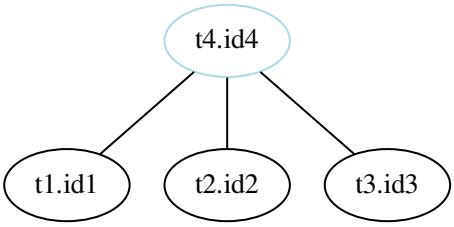

Figure 2.7: Join star pattern with table4 as the center

In this example, if a table is in less than 4 partitions, then in order to use the direct directive (for distributed hash join execution as described in D7.1), a query which repartitions the appropriate table must be provided first. If one table lies in only one partition, then it is automatically being replicated to each node. For the execution of the join graph in each node, we rely on the underlying sqlite database. The query optimizer generates all these possible plans and aided by the estimator, it determines how the given join graph will be executed.

2.2 Query Decomposition and Multi-Query Optimization

This section describes the query decomposition component, whose target is to decompose the query in such a way that for each part we can ensure that the necessary data are properly partitioned, so as to guarantee the correct execution of each part. We will refer to the component of ADP responsible for this operation as query decomposer. Query decomposer takes as input the SQL query from the query transformation module and produces a sequence of queries in the ADP dataflow language that has been described in D7.1. During this process, the decomposer calls the component described in Section 2.1, in order to take some cost estimates and determine the final form of the resulted queries. The exact way that the query decomposer operates depends on the mode of operation. We will define three modes of operation:

1. The normal mode
2. The federated mode
3. The multi-query optimization mode

Before explaining each different mode of operation, we will briefly describe the typical form of the initial queries that are taken as input from the query transformation module and the first steps of the decomposition, which are common for all three modes.

The input query is usually a union query, where each part of the union is a SELECT-FROM-WHERE SQL query, possibly with a GROUP-BY operation. The FROM part may as well contain some nested queries, and the WHERE conditions may contain some disjunctions. Initially, the query is decomposed in disjunctive normal form, that is a sequence of queries that each table in the FROM part is a table name, i.e. we do un-nesting of nested queries, and in the WHERE part we do not have disjunctions. This way, we may end up having more unions, than what we had in the initial query. It is well-known[20] that this form may lead to replicated execution of some operations, for example the same joins may exist in different disjunctions. We chose though to decompose the query this way, as it is more straightforward to define the partition of the tables for each condition and also as we cope with the problem of optimizing the execution by identifying common subexpressions in a complete manner, with respect also to the unions in the initial query, in the multi-query optimization mode. We can now proceed to describe each mode of operation.

2.2.1 Normal Mode

In the normal mode, we assume that all the database tables have been imported into ADP and all the processing takes place inside the system. Each subquery from the initial decomposition into disjunctive normal form is optimized independently. First, common optimization reordering of operators takes place, that is pushing of projections and selections in order to be executed as soon as possible and thus obtaining a smaller size of intermediate results. After that, for each query we have only to face the join ordering problem. The component described in the previous section is called to provide us with the best solution to this problem. One important heuristic that is used in this step is the grouping of joins that can be executed together as star joins, something also provided from this component.

For example, consider the following fragment, which is part of a larger UNION query, and is taken from the OBDA benchmark on the NPD database[5], slightly modified for readability reasons.

```
select
    QVIEW1.‘wlbNpdidWellbore‘,
    QVIEW3.‘cmpLongName‘,
    QVIEW2.‘wlbCompletionYear‘,
    QVIEW7.‘wlbTotalCoreLength‘
from
    wellbore_development_all QVIEW1,
    wellbore_shallow_all QVIEW2,
    company QVIEW3,
    wellbore_exploration_all QVIEW4,
    wellbore_core QVIEW5,
    wellbore_npdid_overview QVIEW6,
    wellbore_core QVIEW7,
    wellbore_core QVIEW8
where
    QVIEW1.‘wlbNpdidWellbore‘ IS NOT NULL AND
    QVIEW1.‘wlbWellboreName‘ IS NOT NULL AND
    (QVIEW1.‘wlbNpdidWellbore‘ = QVIEW2.‘wlbNpdidWellbore‘) AND
    QVIEW2.‘wlbCompletionYear‘ IS NOT NULL AND
    QVIEW3.‘cmpNpdidCompany‘ IS NOT NULL AND
    QVIEW3.‘cmpLongName‘ IS NOT NULL AND
    (QVIEW3.‘cmpLongName‘ = QVIEW4.‘wlbDrillingOperator‘) AND
    (QVIEW1.‘wlbNpdidWellbore‘ = QVIEW4.‘wlbNpdidWellbore‘) AND
    (QVIEW1.‘wlbNpdidWellbore‘ = QVIEW5.‘wlbNpdidWellbore‘) AND
    (QVIEW1.‘wlbNpdidWellbore‘ = QVIEW6.‘wlbNpdidWellbore‘) AND
    QVIEW5.‘wlbCoreNumber‘ IS NOT NULL AND
    (QVIEW5.‘wlbCoreNumber‘ = QVIEW7.‘wlbCoreNumber‘) AND
    (QVIEW1.‘wlbNpdidWellbore‘ = QVIEW7.‘wlbNpdidWellbore‘) AND
    QVIEW7.‘wlbTotalCoreLength‘ IS NOT NULL AND
    (QVIEW5.‘wlbCoreNumber‘ = QVIEW8.‘wlbCoreNumber‘) AND
    (QVIEW8.‘wlbCoreIntervalUom‘ = ’[m   ]’) AND
    (QVIEW1.‘wlbNpdidWellbore‘ = QVIEW8.‘wlbNpdidWellbore‘) AND
    ((QVIEW7.‘wlbTotalCoreLength‘ > 50) AND (QVIEW2.‘wlbCompletionYear‘ >= 2008))
```

For the base table QVIEW8, the following subquery will be produced:

```
distributed create temporary table t8 to 4 on QVIEW8_wlbCoreNumber as
direct select
  QVIEW8.wlbCoreNumber as QVIEW8_wlbCoreNumber,
  QVIEW8.wlbCorelntervalUom as QVIEW8_wlbCoreIntervalUom,
  QVIEW8.w1bNpdidWellbore as QVIEW8_wlbNpdidWellbore
from wellbore_core QVIEW8
where QVIEW8.wlbCorelntervalUom = ’[ft ]’;
```

2.2.2 Federated Mode

In the federated mode, we assume that the tables are stored in external databases and for each query we push as much processing as possible into the corresponding databases. Here, for each external data source, also called endpoint in this context, apart from projection and selection pushing, we also send for execution all the conditions of the WHERE clause that refer to tables only from this endpoint. For example, unary conditions such as column IS NOT NULL, will always be pushed in the corresponding endpoint. This will also be the case for binary conditions that have a constant as one of their operands. For joins, we examine if both operands are coming from the same endpoint and if so, we also push the join in the endpoint. This way, from each endpoint of the subquery we obtain an intermediate result. Then, for the remaining join conditions, i.e. join conditions that have operands coming from different endpoints, we perform the operations inside ADP. For the final ordering of these joins, we also have to ask the component described in Section 2.1.

In order to communicate, execute queries, and take back intermediate results from external databases, special virtual table operators (see D7.1 for the usage of virtual table operators from the system) have been implemented and integrated into the system. A specific operator has to be implemented for each database vendor or other data source that can be used. For example, the system now has implemented operators for communication with MySQL, PostgreSQL and Oracle databases, but it has also operators for importing data from other data sources, like CSV files and even HTML tables. One can easily implement other operators that import data from other sources.

An important aspect of the federated execution is that before sending the queries to remote sources, the system uses the component described in Section 2.3, which, transparently to the decomposer, returns the result, by either executing the query on the remote system, or, in the case that data are available inside ADP, by returning them immediately and avoiding any remote execution.

In both, normal and federated mode, each subquery is treated independently. This has the disadvantage that many optimization opportunities are precluded, especially in complex queries with many unions and nested queries. To deal with this problem, in the next section we present the multi-query optimization mode of our system.

2.2.3 Multi-Query Optimization Mode

During the course of the Optique project it has become evident that each SQL query, obtained by the query transformation module, contains many common subexpressions in different unions or nested queries. To deal with such queries efficiently, it is important to incorporate well known multi-query optimization techniques into the distributed query execution module, adapted to a distributed environment.

Multi-query optimization has been a subject of research for a long time in database community. Early works put effort in finding an optimal global plan, choosing from a set of possible plans for each query, where each plan is decomposed in some tasks, with common tasks that may appear in plans for different queries. To find the optimal global plan, one has also to decide about which common tasks will be chosen for reuse, since the materialization of the temporary task result may not always be more efficient than the recomputation of the result. [35, 36] present a formulation of the problem as a search problem and use the A* algorithm to find the optimal solution, paying attention to the choice of a good heuristic function, whereas [31] employs a dynamic programming algorithm to deal with the problem. Subsequent work[34, 39] use transformation-based optimizers[13, 12] to represent all queries in the same query tree and identify the same logical expressions. These methods offer solutions to both the problems of identifying possible common subexpressions and also to deciding for which of them it would be gainful to materialize. Apart from that, these methods inherit the advantage of extensibility from the transformation-based optimizers.

We chose to build on the work presented in [34], as we can easily model partitioning of tables and conditions required for a join or other operations in a distributed environment, as enforcers in the physical query DAG described in this work. The exact modifications that were needed in the construction of the DAG and the search algorithm depend heavily on design choices and implementation techniques of ADP. We will present these modifications briefly, as the exact version of the search algorithm as well as a detailed comparison between different algorithms with different choices for the DAG is still work in progress.

- Each table is either partitioned to a fixed number of partitions, or is replicated in each node of the system as a whole. The choice for the base tables depends on a threshold. Tables with size larger than the threshold are partitioned and tables with size smaller than the threshold are replicated. In the search algorithm, for each intermediate table, we decide to partition it, if at least one of each input tables is partitioned. The usage of an arbitrary number of partitions is left as future work, but as has been shown by the experiments in Section 4.2.2, usually this method gives good results.

- As intermediate results that are transferred through the network are always written to disk, we take as granted that these are always materialized, and we avoid examining if it would be profitable or not to do so. This decision also reduces the search space, as we have to decide only about the input results of operators that can have them readily available locally.

- Another important decision that can be used to reduce the search space is to avoid repartitioning of data as much as possible. At each step that we must choose the next operator, if we can choose an operator that can be executed without repartition, this would be preferable.

2.3 Caching over the network

This section presents an important optimization for the case of federated query execution, where original data is stored in remote databases. Large responses require many round-trips between the system and the remote sources. For this reason, fetching the results of a query over a slow network will be both slow and expensive. On the contrary, there are containers inside the system which are used to store data from remote sources and the speed between them and the system is expected to be higher and less expensive. Therefore the cache mechanism caches the results of previously executed operators for future use. By caching the results of previously requested operators, the system manages to:

- have much faster response for the stored results
- save network bandwidth
- reduce remote source workload

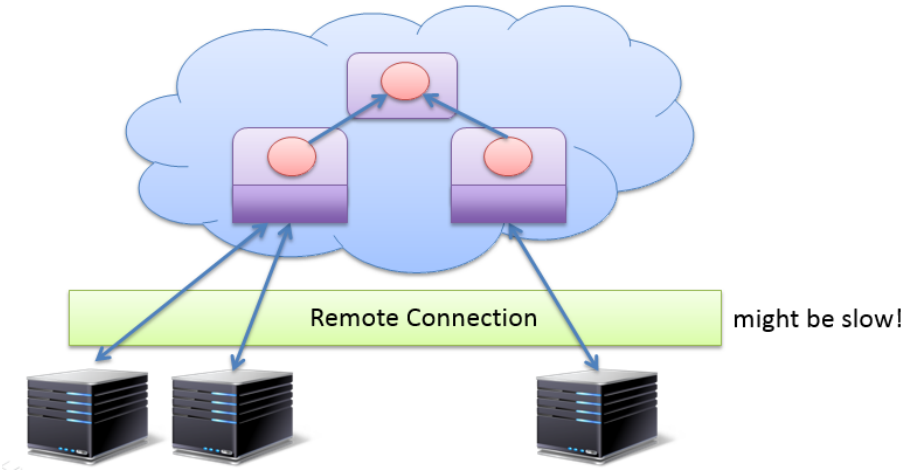

Figure 2.8: Setting of network caching.

2.3.1 Implementation Details

The cache supports a bootstrapping operation. More specifically the module stores all the vital information in disk. Therefore, if the system shuts down, whether normally or not, the cache will be able to restore its operation, by initializing its indexes with the info that has previous stored. By restoring its previously operation setup, it is able to satisfy incoming requests, whose results had been stored in the cache, before the the system shut down. As vital information, we define the data needed by the cache algorithm for the results' replacement.

The cache is responsible to manage the rate with which it sends queries for execution to each remote source. This happens because the remote server workload can become too heavy, as the network congestion can be increased. For this reason, the cache runs a scheduling algorithm, in order to control the amount of data that is processed and transported. Currently, the scheduling policy is very simple. The cache can only send up to a predefined number of queries to each data source. We plan to alter the scheduling algorithm with one that is very close to the TCP congestion control algorithm.

When the results of an operator(which are not already stored in the cache) are requested, the cache will make a request to the corresponding source in order to acquire the operator results. If one or more requests are made for the same operator during the time of the storing, then the cache module will avoid to make the same request again. When the results are ultimately stored, the cache component will inform all the callers about the results' availability.

The cache module allows to users to define a stale threshold. Through the stale threshold, a user informs the system, that he does not want to acquire results which have not been updated within the defined period. The stale threshold may differ among the users.

There is a list of 3 available caching algorithms for use. The first algorithm is the Unlimited Algorithm, which stores the results of all the previously requested operators, without deleting any of them. The second algorithms is the LRU Algorithm. This algorithm removes some of the already stored results, when the cache storage is full, based on how recently the results have been requested for the last time. The last algorithm is the Federated Algorithm. The Federated Algorithm measures the benefit that the system will have from the saving of the results of an operator. When the cache storage will be filled, the cache will keep storing the results which will maximize its total benefit.

In order to define the cost model of the federated cache algorithm, we must first define the following parameters.

- probability of asking for Q_i: $P_Q(Q_i)$

  By storing the number of times that a query has been formulated and the number of total queries formulated, then it can be considered that

  $$P_Q(Q_i) = \frac{\sharp Q_i}{\sum_j \sharp Q_j}$$

- response time for Q_i in the underlying databases: $T_{DBS}(Q_i)$

  It is defined as the average of all the previous response times for that query. This parameter gives us important information about how complex the queries are: whether the answer is big or the communication cost is high, etc.

- response time for Q_i in the cache: $T_{CACHE}(Q_i)$

  It is defined as the average of responses times for that query in the cache.

Based on these parameters, we define the benefit of having a query Q_j in the cache, as follows:

$$G(Q_j) = P_Q(Q_j) * (T_{DBS}(Q_j) - T_{CACHE}(Q_j)) \tag{2.1}$$

Based on the above-mentioned benefit definition, the federated caching algorithm formula is the following one:

$$max_{A \subseteq Q} \sum_{j \in A} G(Qj) \tag{2.2}$$

$$\text{subject to} |A| \leq |Cache| \tag{2.3}$$

where Q is the set of the queries, $|A|$ is the size of the results of the queries in the set A in bytes and $|Cache|$ is the total size of the cache.

It is important to be noted that neither the LRU Algorithm nor the Federated Algorithm will delete results, which are currently in use by one or more users.

2.3.2 Evaluation

For the evaluation of the cache's benefit, a demo was created which evaluates the total execution time. Based on the demo setup, there is a list of 20 discrete queries and 4 clients, each of one would make 30 sequential request to the cache. The query requests follow the Zipf distribution and the Zipf parameter differs among the clients. While the Zipf parameter differs among the clients, however it remains the same for each execution run. The stability of the Zipf parameter among the execution runs, along with the stable seed that is used in order to generate queries, ensures the execution of all the experiments under the same conditions.

There are two parameters which are used in order to evaluate the cache performance. The first parameter is the replacement algorithm and the second one is the size of the cache. The evaluated replacement algorithms are the LRU and the Federated Algorithm. As baseline is defined the total execution time of the above demo setup, when there is no cache.

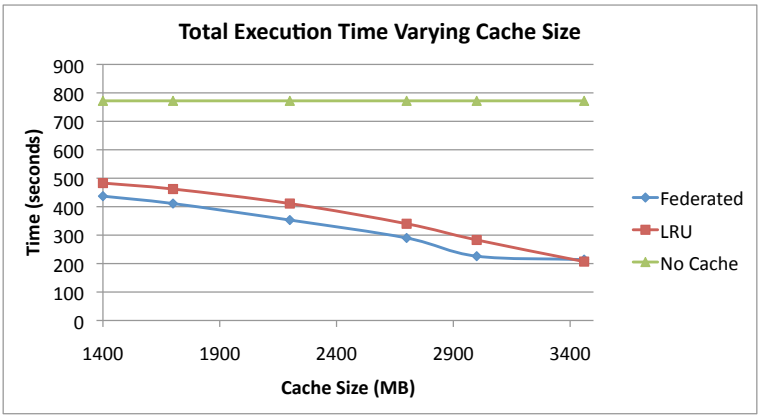

Figure 2.9: Experimental evaluation of network caching

As it was expected, the total execution time of both the replacement algorithms is far more better than the one when there is no cache. Now regarding the two replacement algorithms, we notice that the total execution time of the federated algorithm is lower than the LRU's one. This happens because the Federated Algorithm takes into consideration not only how often a query is requested, but also how complex(or large) the query is. Finally, we observe that as the cache size is getting smaller, the difference of the execution time between the two algorithms is getting smaller.

2.4 Near Interactive Analytics

Modern analytics, like the one we encounter in Optique, face the need to process large amount of data using ad-hoc queries, possibly featuring complex user-defined functions (UDF) that do not come from a pre-defined set of operators with well known semantics. Supporting UDFs is very important because SQL alone is not sufficient or efficient to use in many cases. Furthermore, users require near-interactive response times on complex analytical queries [8, 27].

It has been shown that, in many cases, tree execution plans can answer queries on trillions of objects in seconds [27]. Figure 2.10 shows the generic form of the execution plan of these types of queries. At the leafs of the tree, the data is partitioned in an appropriate way suitable for the application (e.g., partition the documents). Each of the circles in the internal nodes represent operators (e.g., group by) and the connections between them represent data dependencies. The first level of operators (level L_0) perform filtering and joins. The internal operators of the tree (levels L_1 to L_{n-2}) perform partial aggregations. Finally, the root operator (level L_{n-1}) performs global aggregations to produce the final result.

Analytical queries with heavy aggregations appear in several settings. Here we discuss two major categories: data warehouses and NoSQL systems.

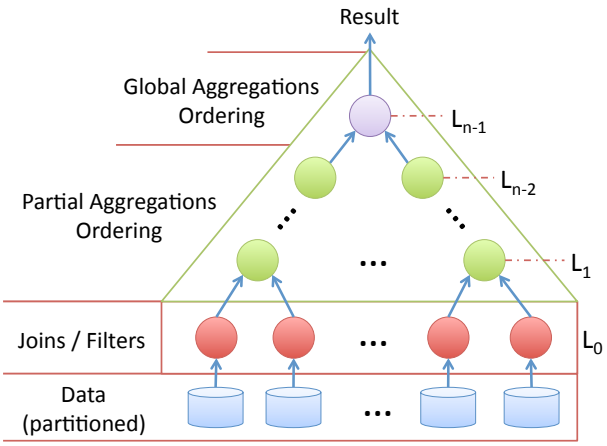

Figure 2.10: Generic form of tree execution plans.

i) Data Warehouses store historical data used to create management reports [37]. Typical queries perform joins and heavy aggregations, and usually return only heavy hitters [33] (the top records sorted by some columns). The following query shows an example expressed in SQL inspired from the TPC-H benchmark [1]:

```
SELECT year, country,
       sum(l_extendedprice) as revenue,
       Summary(l_extendedprice) as report
FROM lineitem, supplier, nation
WHERE l_suppkey = s_suppkey
  AND s_nationkey = n_nationkey
GROUP BY year, country
ORDER BY year, country;
```

The query joins three tables, aggregates the results for each country by year, and computes for each group the revenue. Furthermore it uses the `Summary` UDF to generate a report based on the revenue of each group.

The typical schema of a data warehouse is star or snowflake [28] and is heavily denormalized for performance. The fact table (`lineitem` in the example), is very large compared to the rest of the tables. The typical data placement is to partition the fact table horizontally and replicate all other tables to the locations of the partitions. This way, all the joins of the query are local to each machine and the aggregations are executed in parallel.

ii) NoSQL systems provide techniques to store and process data that is typically in the form of key-value pairs, graphs, or documents [9, 25, 26, 30]. The typical queries involve filtering and transformations using a single input and usually joins are avoided because they are expensive[1]. The following dataflow shows an example of intrusion detection analysis on server logs expressed in FlumeJava [6]:

```
PCollection<String> input = ReadInput("log.txt");
// Parse and convert to log entry objects with
// request IP as keys
PCollection<KV<IP, LogEntry>> entries =
    input.parallelDo(new LogTransform());
PTable<IP, Collection<LogEntry>> groups =
    entries.groupByKey();
// Perform analysis on each group
```

[1] Joins can be implemented on top of these systems [29]

```
PTable<IP, Report> result =
    groups.combineValues(new IntrusionAnalysis());
FlumeJava.run();
```

The dataflow reads the input file "log.txt" (one row/line) and converts it to key-value pairs using the `LogTransform` UDF with the IP of the client as key. After, it groups the entries by IP and performs an intrusion detection analysis on each group using the `IntrusionAnalysis` UDF.

The typical data placement is to partition the files in blocks of fixed size and place them to different VMs using a distributed file system [11]. In the example, `LogTransform` is executed in parallel on each block of the file and `IntrusionAnalysis` is executed in parallel on each group.

iii) Federated Databases are "meta"-database management systems that integrate several databases into a single logical database [4, 10]. The integration is performed using LAV or GAV mappings [22]. The queries are decomposed into several partial queries that are executed to each of the relevant participating databases. The results from the individual databases are joined and aggregated in parallel, typically using bushy trees [15].

Proposed solutions are not sufficient for our cloud setting because they are not elastic or target performance, i.e., evaluate queries as fast as possible, typically trying to minimize the monetary cost or ignore it completely. In a cloud environment, the elasticity and the trade-offs of performance and monetary cost of resources are very important and should be taken into account in a unified approach. To the best of our knowledge, none of the proposed solutions is suitable for the types of analytical queries and the setting we target. We perform an extensive experimental evaluation and compare with Cloudera Impala. Our results show that our approach is very efficient and offers near-interactive response times.

2.4.1 Preliminaries

Data Model and Layout

The tables are horizontally partitioned (using hash partitioning) and stored in a distributed file system on different nodes of the system. The physical layer stores each partition in a different SQLite file and possibly on different containers. The system is mainly optimized for data intensive analytical workload. ADP also supports partial indexes that are build on each partition independently. In the current version of ADP, we use the local disk of each machine to store the partitions.

We partition the tables such that the joins (if any) are local to each machine and only the aggregations are performed in the internal nodes of the tree. We decide the partitioning based on the foreign keys of the tables that indicate the join columns. If the database has only one table (this is the usual case in NoSQL systems, e.g., server logs), then it can be partitioned randomly into parts of equal size. If the database has more than one tables (e.g., data warehouse) the large tables (e.g., fact table) are partitioned on their foreign keys and we replicate all others. For example, for the TPC-H benchmark, we partition the two largest tables (lineitem and orders) using hash partitioning on their foreign key (orderkey).

We assume *updates* are performed in batches periodically (every day or week). Depending on data partitioning, updates can create new partitions (random partitioning) or update existing ones (hash or range partitioning). Each update creates a new version of the table partitions that are changed and the old ones are invalidated [2]. Queries issued after the update, are executed on the updated versions. The old versions are deleted after the currently running queries have finished executing.

Aggregate Functions

We support queries with aggregate functions that are **distributive** or **algebraic** [17]. The distributive aggregate functions can be computed in a distributed fashion because they are both commutative and associative: $f(a, b) = f(b, a)$ and $f(f(a, b), c) = f(a, f(b, c))$.

The algebraic aggregate functions can be computed by applying an algebraic transformation to the results of a set of distributive functions. An example of an algebraic function is AVG, since it can be computed

using the SUM and $COUNT$ distributive functions. All pre-defined aggregate functions in SQL are either distributed or algebraic functions. Let T be a table with two partitions T_1 and T_2. The following hold for SQL:
$f(T) = f(f(T_1) \cup f(T_2)), f \in \{first, last, max, min, sum\}$
$count(T) = sum(count(T_1) \cup count(T_2))$
$avg(T) = sum(T)/count(T)$

We can define more complex algebraic functions using existing functions. An example is standard deviation (stdev) that can be computed as:

$$\sigma_S = \frac{1}{count(S)} \cdot \sqrt{count(S) \cdot sum(S^2) - sum(S)^2}$$

with S being a set of numbers.

In addition, the functionality of the system can be extended by registering UDFs that can have arbitrary code as distributive functions. An example is the reservoir sampling [38] that selects a subset of the records from a particular table randomly with equal probability.

Queries

A **query** is expressed in SQL and can have one or more tables combined with joins, filters, and group by using distributed or algebraic functions. All the algebraic functions are transformed into transformations of distributive functions performing the appropriate expansions and re-writings. We present an example to illustrate. Assume the following query on two tables `R` and `S` that are partitioned on `r` and `s` columns respectively:

```
select avg(a) as avg_a from R, S where r = s;
```

To construct the execution tree, each query is transformed into four conceptual queries: *leaf*, *internal-init*, *internal-recursive*, and *root*. The **leaf query** contains the filters and joins as follows:

```
select a, b from R, S where r = s;
```

The **internal-init** query contains the initialization part of the distributive functions and the **internal-recursive** query contains the recursive part of the distributive functions that compute the partial results. Notice that this is where the commutative and associative properties are needed: the functions can be executed in any order and the result will be correct. The same query is used for all internal levels. The two queries of our example as defined as follows where we observe use of *count* and *sum* to compute the *count* function:

```
// Internal-Init:
select sum(a) as sum_a, count(a) as count_a
from leaf;

// Internal-Recursive:
select sum(sum_a) as sum_a, sum (count_a) as count_a
from intern_init;
```

The **root** query contains the algebraic functions to compute the global aggregations using the partially aggregated results from the internal query as follows:

```
select (sum(sum_a) / sum(count_a)) as avg_a
from intern_rec;
```

Notice that if the query does not contain algebraic functions, the root query is identical to the internal-recursive query. For efficiency, in practice we merge the *internal-init* and *leaf* into a single query.

The query is transformed into an execution plan with operators. An **execution plan** is modeled as $tree(ops, flows)$ with nodes (ops) that correspond to leaf, internal, or root queries and edges ($flows$) that correspond to data dependencies between the nodes. An **operator** in ops is modeled as $op(cpu, memory, disk, time)$,

with *cpu* being the CPU utilization, *mem* being the maximum memory needed for its normal operation, *disk* being the disk resources, and *time* being the execution time of the operator in isolation. These estimations can be either computed or collected by the system at runtime [24]. A **flow** between two operators, producer and consumer, is modeled as $flow(producer, consumer, data)$, with *data* being the size of the data transferred from the producer to the consumer. A **schedule** S_T of an execution plan T is a set of assignments of its operators to containers, i.e., $S_T = \{(T.ops[1], c_1), (T.ops[2], c_2), ...\}$, with c_1 being the container that operator $T.ops[1]$ is assigned to.

2.4.2 Experimental Evaluation

The objectives of the experiment are: *A)* evaluate our methodology showing that we can achieve near-interactive response times for analytical queries and compare with Cloudera Impala, and *B)* show that we can efficiently execute complex analytical queries with UDFs that have arbitrary user code. We present each of these in the following paragraphs.

Experimental Setup

Execution Environment: The resources used for our experiments were kindly provided by Okeanos[2], the cloud of GRNet[3]. We used up to 64 VMs, each with 1 CPU, 4 GB of memory, and 20 GB of disk. We measured the network speed to be around 150 MBit.

Systems: We compare our approach with the latest version of Cloudera Impala [8], that is the state-of-the-art in-memory analytics platform.

Datasets: We use two datasets: the TPC-H [1] that is modeled after typical data warehouse settings and the *Freebase* dataset[4]. The TPC-H benchmark has eight tables:

$lineitem(128, l\_orderkey), orders(128, o\_orderkey), part(1),$
$partsupp(1), supplier(1), customer(1), region(1), nation(1)$

In parenthesis we show the number of partitions we created for each table and the key based on which the partitioning was performed. We partition tables *lineitem* and *orders* on their foreign key using hash partitioning and replicate the other tables. We used the 64GB and 128GB scale factors. Figure 2.11 shows the sizes of the tables. We observe that *lineitem* is very large compared to the rest of the tables.

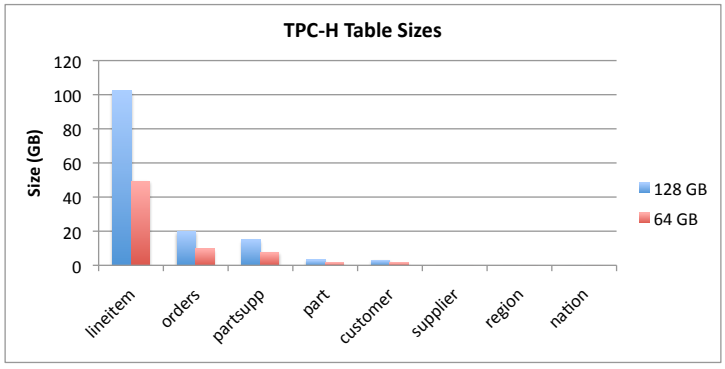

Figure 2.11: TPC-H table size distribution

The *Freebase* dataset contains approximately 2.5 billion triples in the form of N-Triples RDF[5]: <*subject*> <*predicate*> <*object*> "." and its size is 250 GB. If the object is text, it is tagged at the end with the language symbol (e.g., @en means the text is in english). We load the data into one table with three columns.

[2] okeanos.grnet.gr
[3] www.grnet.gr
[4] https://developers.google.com/freebase/data
[5] http://www.w3.org/TR/rdf-testcases/#ntriples

Queries: We use a subset of the TPC-H queries that cover a wide range of the types of queries we target. We choose queries 1, 3, 4, 5, 7, and 9. Q1 uses only table *lineitem* and 8 aggregate functions. Queries 3 and 4 have small number of joins (less than 3) and small number of aggregate functions. Queries 5, 7, and 9 have large number of joins and several aggregate functions.

For the *Freebase* dataset, we used two queries with complex UDFs to create a histogram of the languages that appear in the dataset. The *first* query uses regular expressions to separate the language of the object and then counts the languages. The query is defined as follows:

```
select lang, count(lang) as c
from (select regexpr('.*@(.*)', o) as lang
      from freebase
      where o like "%@%")
group by lang
order by c desc;
```

The *second* query uses reservoir sampling to select from the table 1 million rows and compute the histogram using a UDF that detects the language of a given text using a statistical model. The query is defined as follows:

```
select lang, count(lang) as c
from (select detectlang(lang) as lang
      from (select sample(1000000, lang) as lang
            from freebase)),
group by lang
order by c desc;
```

Near-Interactive Analytics

In our first set of experiments, we measure the efficiency of the service by executing a single query at a time and measure their execution times. We use the TPC-H benchmark with 64 VMs at Okeanos and a 3-level execution tree. We compare with a previous version of ADP that uses graphs to execute queries. Figure 2.12 shows the results.

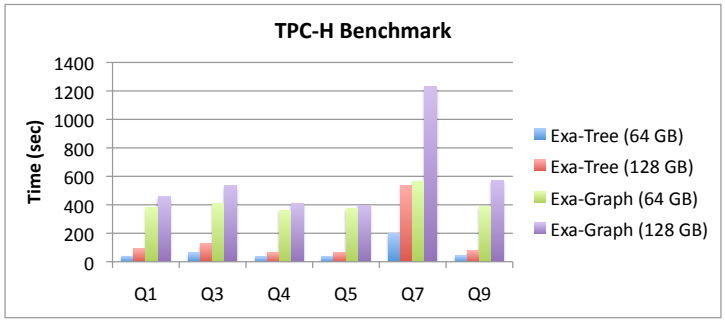

Figure 2.12: TPC-H queries using graph and tree execution plans on 64 containers.

We observe that queries executed with the tree execution plan run significantly faster. The main reason is the tree abstraction in combination with the exploitation of data partitioning. The basic functionality of ADP uses a lattice (all-to-all connections) to partition the data and perform the aggregations in parallel. With the tree abstraction, we radically reduce the number of connections making the system to perform up to an order of magnitude better and offer near-interactive response times (40 sec on average).

Figure 2.13 shows the comparison with Impala using 64 GB of data on 64 VMs. We observe that ADP with the tree abstraction is very efficient for the types of queries we focus in this work. This experiment shows that our service is able to compete with state-of-the-art systems. The main reason for this performance is the data partitioning exploitation that does not produce a lot of network traffic. Since Impala runs entirely in memory, we were not able to run the queries for the 128 GB scale and query 9 for the 64 GB scale.

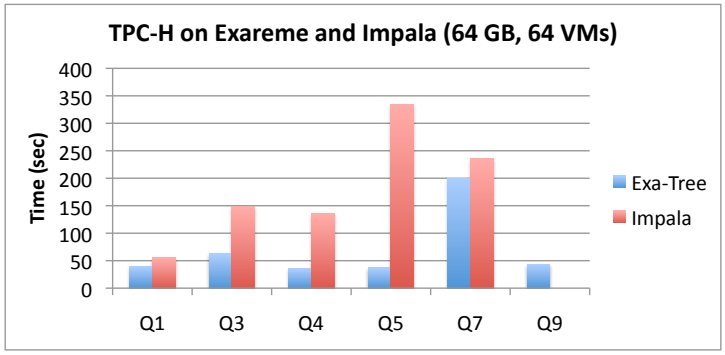

Figure 2.13: TPC-H queries with 64 GB of data on Impala and ADP using 64 containers.

Complex Analytics

One of the advantages of our approach is the support for complex UDFs with arbitrary user code. In this experiment, we use the *Freebase* dataset to create a histogram of the languages using 64 VMs with the two queries discussed earlier. Table 2.1 shows the results. We observe that the histogram computed using the sampling is proportionally similar to the one computed using the entire input.

Table 2.1: Freebase Languages

All (2.4B)		**Sample (1M)**	
lang	**count**	**lang**	**count**
en	134096634	en	115335
fr	28091737	fr	23991
de	27890842	de	23906
es	26934217	es	23462
it	26516667	it	23148
...	...	...	...

The execution time of the queries is shown in Figure 2.14. The first query that uses regular expression, was executed in 1966 seconds. The operators at the leafs of the execution tree took most of the time since computing 2.4 billion regular expressions is expensive. The second query that uses reservoir sampling finished in 339 seconds, with similar results. Random sampling is a standard method to reduce data and execute aggregate queries with good approximations.

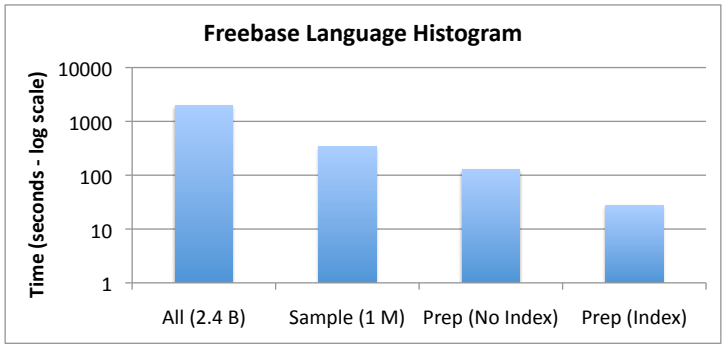

Figure 2.14: Freebase execution time.

We also did a pre-processing of the object column by extracting the language tag and creating an additional column on the table. In this case, the histogram on the whole dataset is computed in 107 seconds without indexes and in 27 seconds using indexes.

2.4.3 Related Work

Data Warehouses: Data warehouses store very large volumes of data (in the orders of petabytes) and are typically used for reporting and historical analysis to discover trends. Several systems have been implemented that are open source (e.g., Hive [37]), proprietary (e.g., Tenzing [7]), or commercial (e.g., Vertica [21]).

The most popular open source warehouses are based on MapReduce [9, 37] and typically offer high level languages (SQL) to express queries that are transformed to one or more MapReduce jobs to execute them. The MapReduce abstraction is not efficient for the heavy aggregate queries we target. In MapReduce, the multi-level aggregations can be expressed with multiple jobs, however the tree abstraction is more efficient. In addition, the optimization goal of these systems is to minimize the number of MapReduce jobs they produce, while at the same time, to maximize parallelization in order to minimize the total execution time. The monetary cost of the resources is by and large ignored.

Dremel [27] and Cloudera Impala [8] are very efficient with aggregate queries executing them as trees. The optimization goal of these systems is performance and, to the best of our knowledge, are not elastic.

NoSQL Systems: Several systems have been proposed to manage data in format different than relational tables. Examples include Sawzall [32], PigLatin [30], and FlumeJava [6]. These systems are either built on top of MapReduce. These systems are not suitable for the queries we target in this work. The MapReduce abstraction is not efficient for multi-level aggregation queries. Furthermore, there is no clean and simple way to define new UDFs along with their properties and use them efficiently in the optimization process. Finally, the monetary cost is ignored.

Federated Databases

Federated databases integrate several individual databases, each with its own data and schema, into a single logical database [16]. The best way to execute joins in this setting is using bushy trees over the federated sources [15, 23]. To the best of our knowledge, there is no federation system built for a cloud environment that takes into account the monetary cost of using the resources.

Chapter 3

Stream Processing

3.1 Introduction

Data stream processing systems offer the ability to make quick and important decisions. They also offer to ordinary users the ability to view information in real-time. This is because these systems provide low latency, a functionality that is typically not provided by distributed systems that target analytical query workloads. The difference between the static and the streaming model is that the former is used to for queries issued against a specific instance of the database, while the later registers the queries and produces results continuously. This difference is illustrated in Figure 3.1.

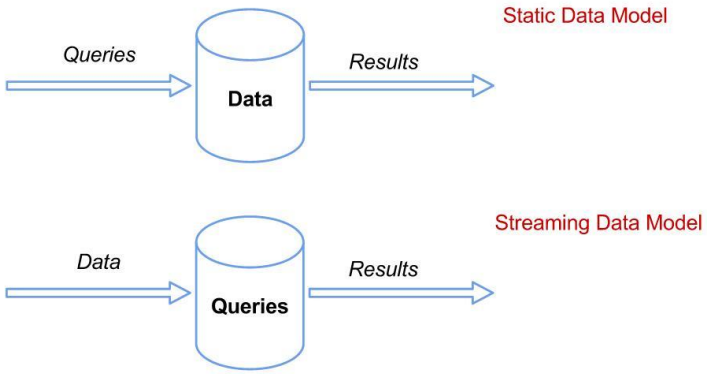

Figure 3.1: Static (top) and Streaming (bottom) models

A stream is an infinite sequence of records ordered by time. To support streaming functionality as a primitive in the engine, we extended the language of the system and implemented streaming versions of relational operators. In the following sections, we present all the modifications we did and the functionality we currently support.

3.2 Illustrative Example

In this section, we present a simple example to show the basic streaming functionality of the system. The following example defines a new stream that connects to a data source:

```
create stream turbine_stream as
  select * from (streamdata 'turbine-stream.csv');
```

where *streamdata* is a streaming data source that simulates the production of infinite data using the data provided by Siemens. The *streamdata* operator is described in more detail in section 3.4.2. Essentially,

streams can be used as tables in queries. The following example selects the records from the stream defined earlier:

```
select * from turbine_stream;
```

The result of this query is as follows:

```
[1|12/05/2010 00:00:00[2|GasTurbine2103/01[3|TC255[4|78.099
[1|12/05/2010 00:01:00[2|GasTurbine2103/01[3|TC255[4|77.599
[1|12/05/2010 00:02:00[2|GasTurbine2103/01[3|TC255[4|77.199
[1|12/05/2010 00:03:00[2|GasTurbine2103/01[3|TC255[4|77
[1|12/05/2010 00:04:00[2|GasTurbine2103/01[3|TC255[4|76.699
...
--- [0|Column names ---
[1|timestamp [2|assembly [3|sensor [4|value
```

where the last line presents the names of the fields. Notice that the stream is infinite and the previous query will never finish execution. Since streams can be treated as tables, we can use operations such as WHERE (filtering), GROUP BY (aggregation), UNION, and JOIN. For example, we can use the *where* clause to filter records from the input stream:

```
create stream turbine_stream as
  select * from (streamdata 'turbine-stream.csv');

select * from turbine_stream where value < 78.0;
```

The result of this query is:

```
[1|12/05/2010 00:01:00[2|GasTurbine2103/01[3|TC255[4|77.599
[1|12/05/2010 00:02:00[2|GasTurbine2103/01[3|TC255[4|77.199
[1|12/05/2010 00:03:00[2|GasTurbine2103/01[3|TC255[4|77
[1|12/05/2010 00:04:00[2|GasTurbine2103/01[3|TC255[4|76.699
...
--- [0|Column names ---
[1|timestamp [2|assembly [3|sensor [4|value
```

3.3 Stream Operators

We have implemented streaming versions of relational operators in order to deal with infinite streams. These operators are listed below:

- *timeslidingwindow, slidingwindow* operators to express a WINDOW operation,
- *wcache* operation to express JOIN between streams and,
- *streamunion* operation to express UNION between streams

We also needed to implement the following auxiliary operators:

- *ordered* operator, to define that an stream produce ordered by timecolumn or window id records,
- *streamdata* operator, for simulate the stream production,
- *opc* operator, to communicate with OPC servers and get raw streams

To be able to process the records in a streaming fashion, we must specify that each stream is *ordered* in the time dimension. The *ordered* operator indicates that each stream produced is ordered by time (or window id in WINDOW operations). So the previous example becomes:

```
create stream turbine-stream as
  ordered select * from (streamdata 'turbine-stream.csv');

select * from turbine-stream where value < 77;
```

3.3.1 Window Operators

To support window operations, we define two new operators named *slidingwindow* and *timeslidingwindow*. These operators are equivalent to the *time-based* and *tuple-based* window operators defined in [3].

Slidingwindow is a *tuple-based* window operator that creates "blocks" of equal number of records. More specifically, the *slidingwindow* operator takes a number $l \in \mathbb{N}$ parameter, that defines how many records to put in each block. Formally, this operator outputs a relation R consisting of blocks of l tuples of the form (wid, t), where wid is an integer that indicates the index of the window, and t is a tuple that belongs to the original stream (and follows its schema).

The syntax of this operator is presented as follows:

```
 slidingwindow window:<number of tuples>
```

The following query presents an example:

```
select *
from (ordered slidingwindow window:2
  select * from (streamdata 'turbine-stream.csv'));
```

In the query described above, we created a window by using a *slidingwindow* statement. The expression window:2 expressed the length of the window (which is 2 tuples). The result of this query is listed below:

```
[1|0[2|12/05/2010 00:00:00[3|GasTurbine2103/01[4|TC255
[1|1[2|12/05/2010 00:00:00[3|GasTurbine2103/01[4|TC255
[1|1[2|12/05/2010 00:01:00[3|GasTurbine2103/01[4|TC255
[1|2[2|12/05/2010 00:01:00[3|GasTurbine2103/01[4|TC255
[1|2[2|12/05/2010 00:02:00[3|GasTurbine2103/01[4|TC255
[1|3[2|12/05/2010 00:02:00[3|GasTurbine2103/01[4|TC255
[1|3[2|12/05/2010 00:03:00[3|GasTurbine2103/01[4|TC255
--- [0|Column names ---
[1|wid [2|timestamp [3|assembly [4|sensor
```

Here we observe that a new column has been added in the table, which is the id of the window that the record belongs to.

We can apply a GROUP BY (aggregation) combined with a WINDOW operation. Consider the problem to count the number of records that are included in a window. This can be solved by adding the aggregate "COUNT" in the above query, as shown below:

```
select *, count(*)
from (ordered slidingwindow window:2
       select * from (streamdata 'turbine-stream.csv'))
group by wid;
```

Results:

```
[1|0[2|12/05/2010 00:00:00[3|GasTurbine2103/01[4|TC255[6|1
[1|1[2|12/05/2010 00:01:00[3|GasTurbine2103/01[4|TC255[6|2
[1|2[2|12/05/2010 00:02:00[3|GasTurbine2103/01[4|TC255[6|2
[1|3[2|12/05/2010 00:03:00[3|GasTurbine2103/01[4|TC255[6|2
--- [0|Column names ---
[1|wid [2|timestamp [3|assembly [4|sensor [6|count(*)
```

The *time sliding window* operator defines a sliding window by assigning a window id to a block of records whose timestamp is inside a time range. This deviates from the definition of the *time-based* window operator defined in [3]. This operator, for every $f \in \mathbb{N}$ time units, and for a given stream S, outputs tuples of the form (wid, i, t), where t is a tuple that follows the schema of the input stream, wid is the index of the window, $t \in S$, and $i \in \mathbb{N}$ the index of the timestamp inside the window, and for every tuple (wid, i, t) that belongs to each group, the timestamp τ of this tuple should be within the time interval l that represents the length of the window (and it is also passed as a parameter). In the implementation of this operator, we have taken into account the sequencing methods that have been introduced in STARQL and have been reported in the deliverables D5.1 and D5.2. These sequencing methods have been defined in STARQL to specify which assertions go into the same ABox. In the direction of pushing as much computational effort as possible to the back-end, in order to achieve better performance, we took into consideration the idea behind sequencing methods, by providing the "equivalence" parameter, that can be used optionally by the user. So, for example, by passing `equivalence:floor` or `equivalence:ceil` as parameters, the timestamps whose integer representation in a given time unit (e.g., seconds) have the same *floor* or *ceiling* respectively, will share the same index in the respective output column.

The syntax of this operator is shown below:

```
timeslidingwindow timewindow:<seconds> timecolumn:<column index>
frequency:<number of minutes> <OPTIONAL-PARAMETERS>
```

The required fields are the length of the window in seconds, the index of the timestamp column (starting from 0), the frequency as an integer representing minutes. A user can also optionally use one of the equivalence functions described above.
For example, we assume the following table:

```
2010-12-05T00:00:00+00:00
2010-12-05T00:01:00+00:00
2010-12-05T00:02:00+00:00
2010-12-05T00:02:30+00:00
2010-12-05T00:05:00+00:00
2010-12-05T00:06:00+00:00
```

And the following query:

```
timeslidingwindow timewindow:120 timecolumn:0 frequency:1 equivalence:ceil
 select * from table1
```

The query described above creates a sliding window every minute. The length of the window is 120 seconds. The parameter `timecolumn:0` indicates that the timestamp is the first element of every tuple in the input stream. The results of this query are presented below:

```
[1]0[2]0[3]2010-12-05T00:00:00+00:00
[1]1[2]0[3]2010-12-05T00:00:00+00:00
[1]1[2]1[3]2010-12-05T00:01:00+00:00
[1]2[2]0[3]2010-12-05T00:00:00+00:00
[1]2[2]1[3]2010-12-05T00:01:00+00:00
[1]2[2]2[3]2010-12-05T00:02:00+00:00
[1]3[2]0[3]2010-12-05T00:01:00+00:00
[1]3[2]1[3]2010-12-05T00:02:00+00:00
[1]3[2]2[3]2010-12-05T00:02:30+00:00
[1]4[2]0[3]2010-12-05T00:02:00+00:00
[1]4[2]1[3]2010-12-05T00:02:30+00:00
[1]4[2]2[3]2010-12-05T00:04:00+00:00
```

The first column of the results is the *wid* of each window, the second one is the *index*, and, the last one is the tuple, that consists of a timestamp.

3.3.2 Join Operator

In this section, we describe the JOIN operator used to join two or more streams. Joining streams is not easily defined due to the unpredictable rates at which data is produced at the sources. The *wcache* operator takes his name from "window cache" expression, and as mentioned above, is used to join streams. The following example shows this functionality:

```
create stream stream1 as
  select wid, value
  from (ordered timeslidingwindow timecolumn:0 timewindow:3
                select * from (streamdata 'turbine-stream.csv'));

create stream stream2 as
  select wid, value
  from (ordered timeslidingwindow timecolumn:0 timewindow:3
                select * from (streamdata 'turbine-stream.csv'));

select *
from
  (wcache select * from stream1) as a,
  (wcache select * from stream2) as b
where a.wid=b.wid;
```

In this example, we create two streams named 'stream1' and 'stream2' respectively. The two streams have the some output because they read from the same file. Both produce a window with the same duration of 3 seconds. The result of the previous query is:

```
[1|0[2|78.099[3|0[4|78.099
[1|1[2|78.099[3|1[4|78.099
[1|1[2|78.099[3|1[4|77.599
[1|1[2|77.599[3|1[4|78.099
[1|1[2|77.599[3|1[4|77.599
[1|2[2|77.599[3|2[4|77.599
[1|2[2|77.599[3|2[4|77.199
[1|2[2|77.199[3|2[4|77.599
[1|2[2|77.199[3|2[4|77.199
[1|3[2|77.199[3|3[4|77.199
[1|3[2|77.199[3|3[4|77
[1|3[2|77[3|3[4|77.199
[1|3[2|77[3|3[4|77
...
--- [0|Column names ---
[1|a.wid [2|a.value [3|b.wid [4|b.value
```

The *wcache* operator requires the ordered column (timestamp, wid etc) to be the first column. We can define more complicated queries that involve joins and filters. An example is the following:

```
<stream definitions>
create stream stream1 as ... ;

create stream stream2 as ... ;

select * from
```

```
  (wcache select * from stream1) as a,
  (wcache select * from stream2) as b
where a.wid=b.wid and a.value<b.value and a.value<77.0;
```

The result of the query is:

```
[1|3[2|77[3|3[4|77.199
[1|4[2|76.699[3|4[4|77
[1|5[2|76.5[3|5[4|76.699
[1|6[2|76.3[3|6[4|76.5
[1|7[2|76.099[3|7[4|76.3
[1|8[2|75.8[3|8[4|76.099
...
--- [0|Column names ---
[1|a.wid [2|a.value [3|b.wid [4|b.value
```

The following example shows another way to define a join between two streams:

```
Stream Definitions ....

select * from
  (wcache select * from stream1) as a
  NATURAL JOIN
  (wcache select * from stream2) as b;
```

The results are shown below:

```
[1|0[2|78.099[3|0[4|78.099
[1|1[2|78.099[3|1[4|78.099
[1|1[2|77.599[3|1[4|77.599
[1|2[2|77.599[3|2[4|77.599
[1|2[2|77.199[3|2[4|77.199
[1|3[2|77.199[3|3[4|77.199
[1|3[2|77[3|3[4|77
[1|4[2|77[3|4[4|77
[1|4[2|76.699[3|4[4|76.699
...
--- [0|Column names ---
[1|a.wid [2|a.value [3|b.wid [4|b.value
```

A special category of queries that are typically generated by the STARQL language, have 'exists' and 'not exists' operators. These operators are not easy to handle in the generic case, because relational database is forced to execute the inner query for each input tuple. In streaming mode, this kind of conditions, can only be used with JOIN operations. An example of such query is:

```
create stream stream1 as
  select wid, value
  from (ordered timeslidingwindow timecolumn:0 timewindow:3
                select * from (streamdata 'turbine-stream.csv'));

create stream stream2 as
  select wid, value
  from (ordered timeslidingwindow timecolumn:0 timewindow:3
                select * from (streamdata 'turbine-stream.csv'));

select *
from  stream1 as a
```

```
where not exists(
            select *
            from (wcache select * from stream2) as b
            where a.wid=b.wid and b.value>76.5) and value is not null;
```

In the example described above, there are two basic operations taking place; JOIN and WHERE (filtering). In detail we have a JOIN operation between the inner and outer streams, and a filtering query in outer stream ('value is not null'). This query can be executed in streaming mode, and the result is:

```
6|76.5
6|76.3
7|76.3
7|76.099
8|76.099
8|75.8
...
--- [0|Column names ---
[1|wid [2|value
```

The engine also supports joining multiple streams, as shown in the following example:

```
create stream1 as select * from streamdata( ...);

create stream2 as select * from streamdata( ...);

create stream3 as select * from streamdata( ...);

select * from
  (wcahe select * from
    (wcache
     select * from (wcache select * from stream1) as a,
                   (wcache select * from stream2) as b
     where a.wid=b.wid)) as a,
     (wcache select * from stream3) as b
 where a.wid=b.wid;
```

3.3.3 Union Operator

The union of two or more streams is a fundamental operation. For this, we have implemented the *mergeunion* operator. The union between streams is defined as the union of the individual matching windows without removing duplicate records. The mergeunion operator "puts together" all records with same ordered column (timestmap, window id, etc.), like the 'union all' operator of SQL. Below shown an example:

```
create stream1 as
  select * from (streamdata 'turbine-stream.csv');

create stream2 as
  select * from (streamdata 'turbine-stream.csv');

create stream3 as
  select * from (streamdata 'turbine-stream.csv');

select * from (mergeunion 'stream1,stream2,stream3');
```

and the results:

```
[1|12/05/2010 00:00:00[2|GasTurbine2103/01[3|TC255[4|78.099
[1|12/05/2010 00:00:00[2|GasTurbine2103/01[3|TC255[4|78.099
[1|12/05/2010 00:00:00[2|GasTurbine2103/01[3|TC255[4|78.099
[1|12/05/2010 00:01:00[2|GasTurbine2103/01[3|TC255[4|77.599
[1|12/05/2010 00:01:00[2|GasTurbine2103/01[3|TC255[4|77.599
[1|12/05/2010 00:01:00[2|GasTurbine2103/01[3|TC255[4|77.599
[1|12/05/2010 00:02:00[2|GasTurbine2103/01[3|TC255[4|77.199
[1|12/05/2010 00:02:00[2|GasTurbine2103/01[3|TC255[4|77.199
[1|12/05/2010 00:02:00[2|GasTurbine2103/01[3|TC255[4|77.199
[1|12/05/2010 00:03:00[2|GasTurbine2103/01[3|TC255[4|77
[1|12/05/2010 00:03:00[2|GasTurbine2103/01[3|TC255[4|77
[1|12/05/2010 00:03:00[2|GasTurbine2103/01[3|TC255[4|77
--- [0|Column names ---
[1|timestamp [2|assembly [3|sensor [4|value
```

3.4 Stream Data Sources

One of the most important operations for streaming engines, is the connection to external data sources. In our system, this can be done by implementing new operators. Using the abstraction of operator, has the advantage of hiding the low-level implementation details (server connections etc.) and the end user can focus only on the appliaction logic. In this section, we present in detail the data source operators we support that are the *opc* operator and the *streamdata* operator.

3.4.1 OPC Operator

The OPC protocol is a standard that defines the secure and reliable exchange of streaming data for industrial automation. It is platform independent and ensures the seamless flow of information among devices from multiple vendors. The OPC Foundation is responsible for the development and maintenance of this standard. The OPC standard is a series of specifications developed by industry vendors, end-users, and software developers. These specifications define the interface between clients and servers, as well as servers and servers, including access to real-time data, monitoring of alarms and events, access to historical data and other applications. When the standard was first released in 1996, its purpose was to abstract PLC specific protocols (such as Modbus, Profibus, etc.) into a standardized interface allowing HMI/SCADA systems to interface with a "middle-man" who would convert generic-OPC read/write requests into device-specific requests and vice-versa. As a result, an entire cottage industry of products emerged allowing end-users to implement systems using best-of-breed products all seamlessly interacting via OPC.

Initially, the OPC standard was restricted to the Windows operating system. As such, the acronym OPC was borne from OLE (object linking and embedding) for Process Control. These specifications, which are now known as OPC Classic, have enjoyed widespread adoption across multiple industries, including manufacturing, building automation, oil and gas, renewable energy and utilities, among others. With the introduction of service-oriented architectures in manufacturing systems came new challenges in security and data modeling. The OPC Foundation developed the OPC UA specifications to address these needs and at the same time provided a feature-rich technology open-platform architecture that was future-proof, scalable and extensible.

OPC supports a variety of primitive data types as Integer, Float, String, Boolean Date and many others. Besides this, it supports the special type group, which is a structure of one or more data(primitive or not). In order to read data with the use of the opc protocol we implemented the OPC operator. This operator reads the required values from the remote source and returns a stream of values in a relational format. The relational schema is consisted of the names of the required values, and has an additional columns with the timestamp of each record. The opc operator demands from the user to define the following parameters:

- server ip

- server port
- opc server name
- sample time
- a list of access paths

If the server port is not defined, the operator will automatically fill it with the default opc port, which is the 7766. The sample time indicates the rate which the data will be updated. An access path is intended as a way for the client to provide to the server a suggested data path(e.g. a particular modem or network interface). It indicates how to get the data. Therefore the list of access paths defines the access path for each one of the desired data. The access paths must be paths to an exactly one primitive value, otherwise the operator will throw an appropriate exception.

To better illustrate the functionality, we present an example. Let's assume, that we desire to acquire the pressure and the temperature values that the turbine 1 indicates and the indications will be updated every 4 seconds. The corresponding query will be like the below.

```
select *
from (opc ip:localhost port:7766 server:Matrikoc.OPC.Simulation.1
          time:2
          'Configured Aliases.turbine1.pressure,
          Configured Aliases.turbine1.temperature');
```

An instance of the output of the above query is the below:

```
[1|12/05/2010 00:00:00[2|13.4[3|78.099
[1|12/05/2010 00:00:02[2|12.3[3|78.099
[1|12/05/2010 00:00:04[2|12.3[3|78.097
...
--- [0|Column names ---
[1|timestamp [2|pressure [3|temperature
```

3.4.2 Streamdata Operator

The OPC Protocol is a well defined protocol to communicates with external sources, but in our case we want to experiment with real Siemens datasets that are provided in files with CSV format. For that reason, we have implement the *streamdata* operator. *Streamdata* operator takes as input a CSV file and produce infinitive data iterating around the input file. More specifically the streamdata operator has the following parameters:

- *file*: The path to the CSV file
- *ratio*: The ratio of records/sec

The ratio defines how many records per second the operator produces. For example, a value of 2 produces 2 records every second and a value of 0.5 produces one record every 2 seconds.

Consider the case in which two *opc* operators read records from the same OPC Server at the same time. The result should be the same. To simulate this behavior, we load all records from the CSV file into an circular buffer and calculate the starting position of a connected client as follows:

$$currenttime \mod \frac{buffer\_length}{ratio}$$

This way, the output from two different input operators is going to be the same. The records that produce from streamdata operator has a timestamp field which is matched to the time of their production. Each record returned is tagged with the current time of the server.

3.5 Combine Static and Streaming Data

The engine can support queries that combine streams and static tables. For example, consider the following query:

```
create table static (static_value integer);
insert into static values (1);
insert into static values (100);

create stream stream1 as
  select * from (streamdata 'turbine-stream.csv');

select * from stream1, static where static.static_value > stream1.value;
```

The query joins a static table with a stream. Notice that we do not need to use the *wcache* operator in this case, because there is no JOIN between streams, but a JOIN between a stream and a static table. The results of the query are shown below:

```
[1|0[2|12/05/2010 00:00:00[3|GasTurbine2103/01[4|TC255[5|78.099[6|100
[1|1[2|12/05/2010 00:00:00[3|GasTurbine2103/01[4|TC255[5|78.099[6|100
[1|1[2|12/05/2010 00:01:00[3|GasTurbine2103/01[4|TC255[5|77.599[6|100
[1|2[2|12/05/2010 00:01:00[3|GasTurbine2103/01[4|TC255[5|77.599[6|100
[1|2[2|12/05/2010 00:02:00[3|GasTurbine2103/01[4|TC255[5|77.199[6|100
[1|3[2|12/05/2010 00:02:00[3|GasTurbine2103/01[4|TC255[5|77.199[6|100
[1|3[2|12/05/2010 00:03:00[3|GasTurbine2103/01[4|TC255[5|77
--- [0|Column names ---
[1|wid [2|timestamp [3|assembly [4|sensor [5|value [5|static_value
```

3.6 User Defined Functions

The functionality of the system can be extended by implementing UDFs. One of the goals of the system is to eliminate the effort of creating and using UDFs by making them a first class citizens in the query language itself. The system supports row, aggregate, and virtual table functions. Row UDFs take as input one row with one or more columns and produce one or more values. An example is the UPPER function that takes as input a string and converts its letters to upper case. Aggregate UDFs take as input zero or more rows with one or more columns and produce one or more aggregated values. Standard SQL has several aggregate functions, like SUM and AVG. Notice that the row and aggregate functions are different from the ones used in standard SQL. ADP allows them to produce more than one value. Virtual table UDFs (also known as table functions in Postgresql 5 and Oracle 6) are used to create virtual tables that can be used as regular tables. An example is the FILE UDF. It provides access as tables to files of different types. This way, we can import external files on-the-fly during query execution.

We enhanced the syntax of SQL to easily express data pipelines using UDFs. Standard SQL alone, is not sufficient to support modern applications. However, the relational primitives are very good to express relations and data combinations. We use standard SQL to combine data and process them with UDFs whenever the SQL abstractions are not sufficient or not efficient to use. By making UDFs a first class citizen of the language, we can express very complex dataflows using simple primitives. It is easy to process data from external sources, including files (e.g., CSV, TSV) and remote servers (e.g., HTTP, FTP, web services, databases). ADP offers a rich library of UDFs for I/O (file, FTP, web services), statistics (pearson, etc.), data types (e.g., TSV, XML, JSON), and table (pack, expand). The full list of supported UDFs can be found here: http://code.google.com/p/madis/.

3.7 Implementation Details

We implemented all the above functionality in the ADP database engine. In this section, we discuss some implementation details, present some challenges that emerge in stream processing, and the solutions we propose along with some illustrative examples.

A fundamental extension of a static relational database involve the join between streams. The problem with joins exists because most of the systems do not support the merge join algorithm. Instead the query optimizer of relational databases, in many cases, selects to execute a hash-join or a nested loop algorithm using an index for the inner table. In the latter case, the outer table is scanned and the inner index is probed for each record. This will not work for streams since both tables are infinite.

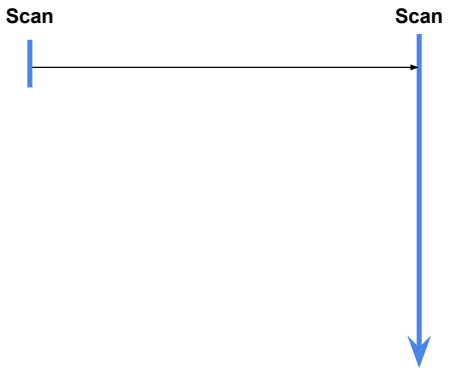

Figure 3.2: Join with static tables

For this reason, we implement the *wcache* operator. Wcache is a virtual table operator that implements all the logic (query optimizer, indexing etc). The base indexes that wcache implements is a scan, and an "window cache" algorithm, which keeps (cached) the next or equal window of records than the value of the questioned. An example is:

```
create stream stream1 as select * from (streamdata ...);
create stream stream2 as select * from (streamdata ...);

select * from (wcache stream1) as a, (wcache stream2) as b
where t1=t2 and a.temperature>10;
```

The query optimizer forces, the engine to make a scan to one of two streams. The stream who performs the scan, probes the values from the other that uses the "window cache". If the probed value "window cached" exists, it retuns all the matches. This way, we have implemented a streaming merge join.

Another challenge is when two operators read from the same input stream. To support this, each stream is hidden behind a buffer. For example consider the following query that performs a self join on a stream:

```
create stream stream1 as select * from (streamdata ...);

select * from
  (wcache select * from stream1) as a,
  (wcache select * from stream1) as b
where a.wid=b.wid;
```

3.8 Rest Interface

In this section, we discuss the ways that clients can interact with the stream processing engine. For this purpose we must specify a type of network communication interaction. There are two basic models, *push*

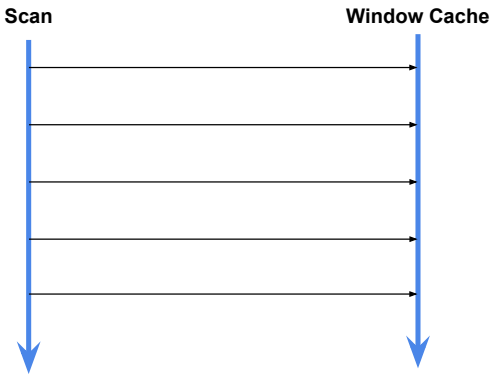

Figure 3.3: Join of two streams.

and *pull*. Each of these models have an active part and a passive part. The two models mainly differ on which part starts the interaction. In the *pull model* the sender waits idle until it accepts a request from a consumer node. Then the sender provides the requested content to the receiver (i.e., the consumer). All that a sender has to do is to keep the content until requested. The main advantage of this model is that the receiving node can choose the content that they want to consume. The disadvantage of this model, however, is that sender may receive too many requests that it cannot serve. This causes redundant iterruptions to the sender. In the *push models*, on the other hand, the sender is the passive part and the receiver is the active. Once the sender has some content to be published/sent to the other node/nodes, it either sends or multicasts the content without having to wait for any preceding request from the receiving end. The advantage of this model is that the sender will not get interrupted while being unable to serve additional requests, but in this case the receivers will get content that is might not interesting for them. For the external communication with the processing engine, we use a push model, because we prefer the receivers to have the flexibility to receive the data that they want to consume.

In more detail, we define and implement a Rest API that enables users to register streaming queries and request the results in a pull fashion. The HTTP Server that implements that REST Interface, interacts continuously with the ADP engine and stores the produced records in the last 30 minutes (this is configurable). This way, multiple clients can issue requests at different time intervals and get the results.

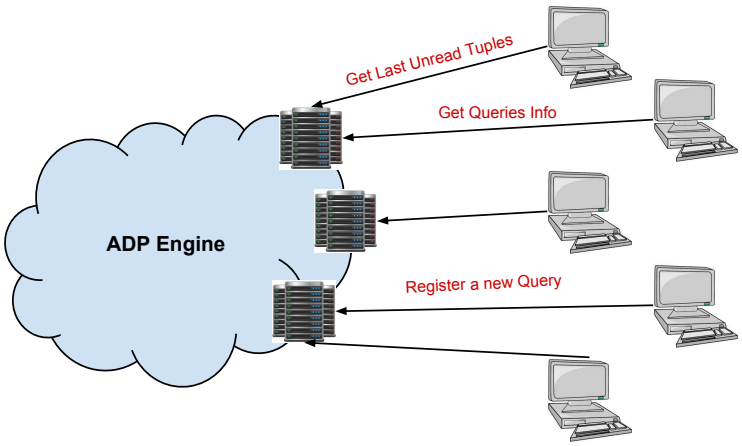

Figure 3.4: High level overview of the stream REST API

The architecture of the system is shown in Figure 3.4. Multiple clients send requests using the REST

protocol. The main functionality of the service is summarized in Table 3.1. Next, we present in detail the REST Interface.

Table 3.1: REST API Functionality Overview

Method	Type	URL	Param	Return
Register	Post, Put	http://HOST/{Name}	register_query = {Query}	200 OK, 409 Conflict Error, 429 Too Many Requests
Delete	Delete	http://HOST/{Name}	-	200 OK, 404 Not Found
Get Info	Get	http://HOST/	-	200 OK
Get Results	Get	http://HOST/{Name}	last={N}, startTimestamp={S}, endTimestamp={E}	200 OK, 404 Not Found

Register a query: The first functionality we support is to register a query. This can be done using the post or put requests and the URL is of the form http://HOST/{Name}, with {Name} being a string that identifies the query. The Rest Server supports only "x-www-form-urlencoded" content-type header. The service returns 200 OK, if all goes well, 409 Conflict Error, if already exists a query with same name. 429 Too Many Requests if the number of registered queries is 5 (for the time being, the maximum number of queries that can be registered is 5). Figure 3.5 shows a screenshot of register query operation. To demo this functionality, we used a browser REST client plugin named "POSTMAN".

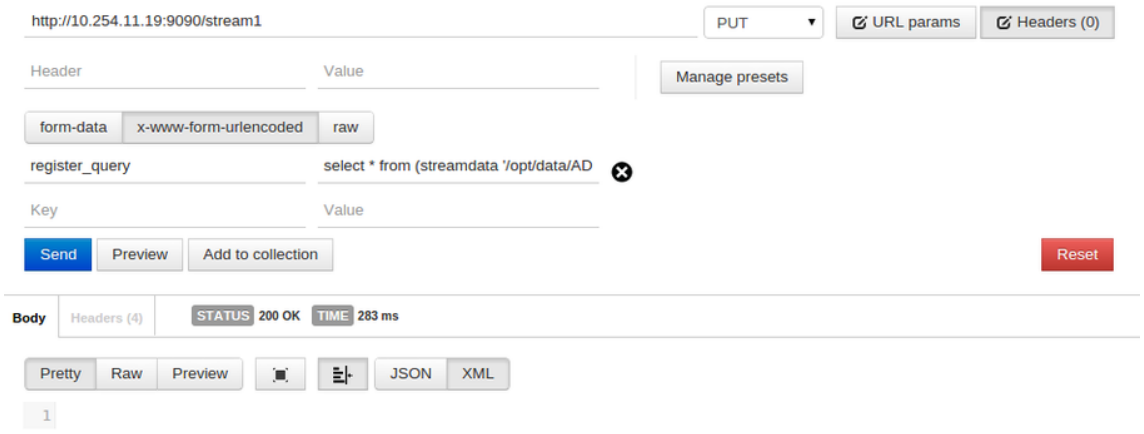

Figure 3.5: Screenshot for a "register a query" operation

Delete a registed query: To delete an already register query we use the delete rest request and URL is of the form http://HOST/{Name}. The service returns 200 OK, if all goes well or 404 Not Found, if not exists query with {Name}. A respective screenshot can be seen in Figure 3.6.s

Get Info: To get information about the registered queries we use the get rest request with the URL http://HOST/. The service return 200 OK, if all goes well. It also returns the names of the registered queries, the date that they were registered, and their current state (Loading, Run, Die) in JSON format. An example of such information is shown in figure 3.7:

Get Results: The Rest Server keeps the records that produced in last 30 minutes. To get the results that produced from an already register query, we use the following URL forms:

i) Get records that were produced in last N seconds:
http://ip:9090/{Name}?last={N}

ii) Get records from the "startTimestamp" and beyond:
http://ip:9090/{Name}?startTimestamp={start} //

iii) Get records between "startTimestamp" and "endTimestamp":
http://ip:9090/Stream Name?startTimestamp={start}&endTimestamp={start}

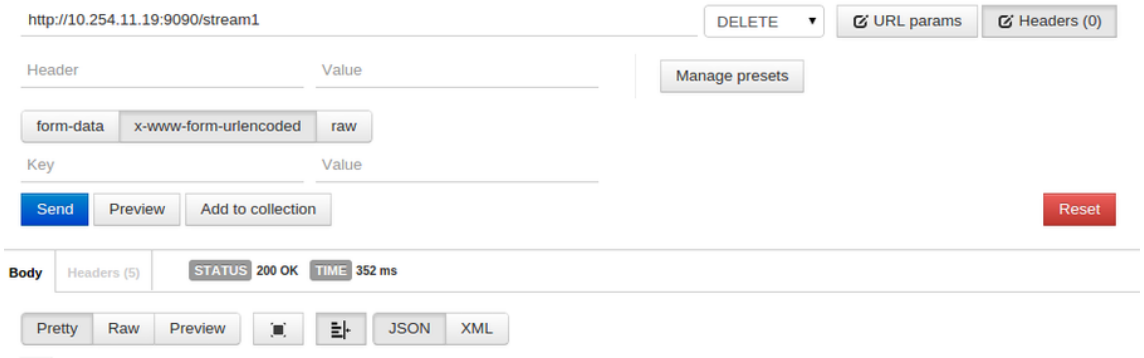

Figure 3.6: Screen-shot for a "delete a registed query" operation

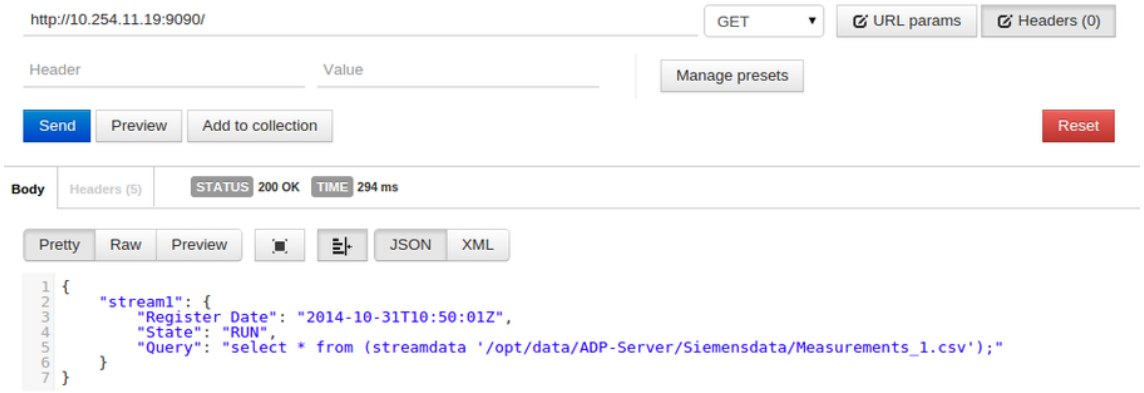

Figure 3.7: Screen-shot for a "get info" operation

The service returns 200 OK, if all goes well or 404 Not Found, if the query with {Name} does not exist. The results are returned in JSON, CSV or a mixed format, declared in the Header Field named "Accept". The "Accept" Header field takes the value "application/json" for JSON format, "text/csv" for CSV format, and "text/json" for JSON format. By default the server sends tuples in mixed format. Mixed format is like CSV with the difference that each tuple is a separate JSON representation, as shown in Figure 3.8:

3.9 Experimental Evaluation

In this section, we present some experiments to measure the efficiency of the streaming operators of the engine and the speedup that JIT techniques provide. The results were obtained executed in a machine with the following specifications: Pentium P6100 double-core CPU at 2 GHz, and 4GB of RAM.

Figure 3.9 shows the execution time of 4 different queries with streaming operators that read from an input stream with a total of 350.000 records. As an input dataset we used one of the datasets provided by Siemens in the context of the project. We observe that the engine is efficient, in terms of query execution time, and that just in time compilation using (PyPy) makes it even more efficient. Figure 3.10 shows the throughput of the same queries. We observe that expensive operators like the join, can be processed at a rate of more than 20000 records per second. These results are promising and indicate than the distributed execution of the streams will be even more efficient.

Also took measurements from a real SQL query generated by a STARQL query. We use a SIEMENS dataset with 420000 rows. In order to measure the query response time of the ADP stream query engine, we saved the results of the *timeslidingwindow* operator in a static table. And then we ran the query in streaming

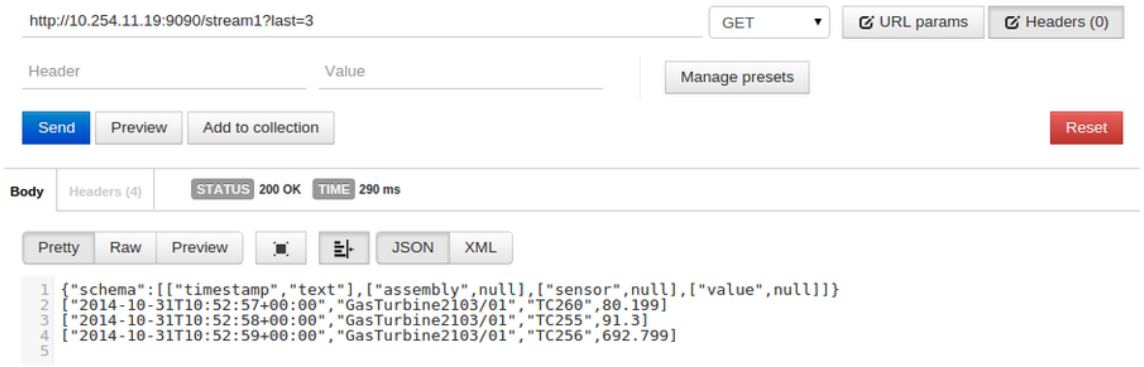

Figure 3.8: Screen-shot for a "get results" operation

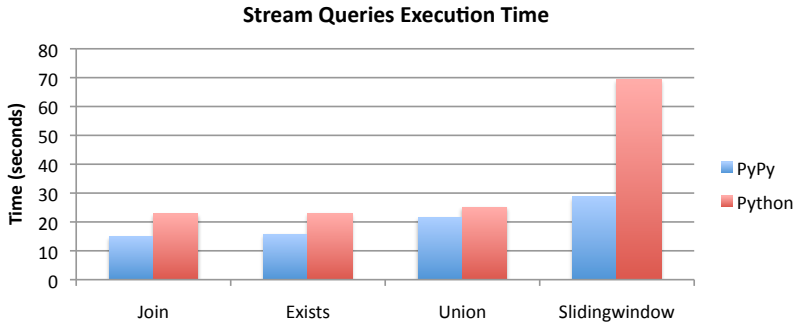

Figure 3.9: Execution time of streaming operators.

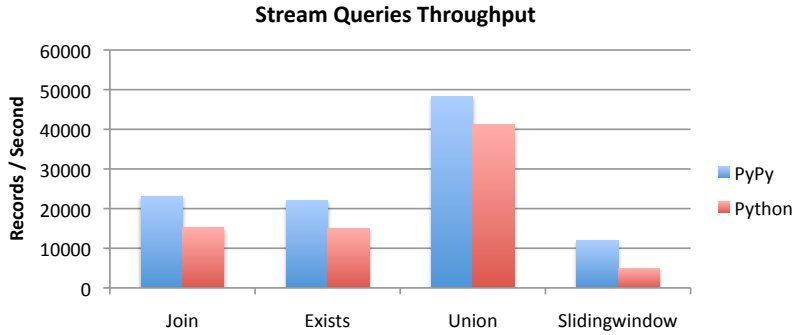

Figure 3.10: Throughput of streaming operators.

and static mode. In static mode we ran the query in two modes: building an index in the precomputed static table and without building an index. Reasonably, the execution of the query that in static mode without index takes significatly more time, as its reponse time is more than 10 minutes. The results are shown in Figure 3.11

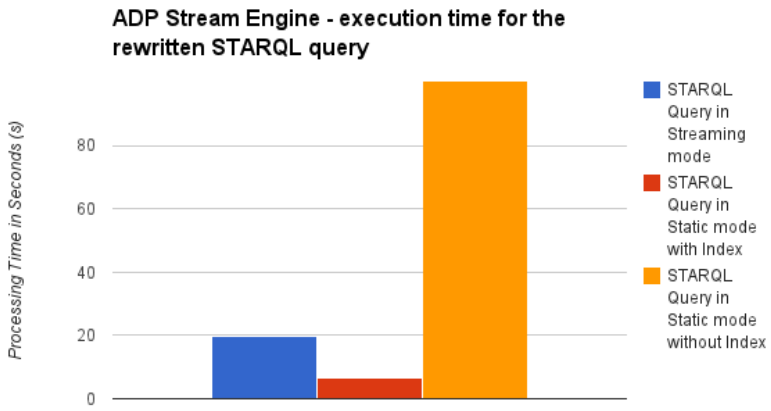

Figure 3.11: Execution time of the translated STARQL Query

Chapter 4

Compare with State-of-the-art Systems

In this chapter, we examine the performa of our system compared to the current leading SQL-engines over Hadoop [1]. Apache Hadoop project became a reliable and scalable system for distributed computing designed to support applications with high-availability using commodity hardware. Additional projects developed on top of Hadoop, exploiting its features and providing various programming models for distributed computing. Our experiments focus on the following query processing engines over Hadoop that provide SQL interface.

Apache Hadoop comes with two basic layers : Hadoop Distributed File System (HDFS) and the Hadoop YARN. The HDFS is designed to provide high-throughput access to application data and fault-tolerance throw the replication policy. Hadoop Yarn is for job scheduling and cluster resource management. On top of Yarn are current exists two application frameworks for defining Yarn-based job(s): the default MapReduce (mr) and the Tez[2]. *Apache Hive* [3] lies on these application frameworks (mr/Tez) in order to convert SQL-like queries to a set of one or more Yarn-based job(s). *Impala*[8] is also an SQL-like engine over hadoop, however is operate completely in memory, aiming to provide real time query processing. Impala is part of the Hadoop ecosystem since its lies on HDFS, Hive Metastore and Statestore for data and metadata.

In the next sections, we describe the basics of each system, our cluster and software configurations, the experiment workload and results.

4.1 Large Scale Experiments

4.1.1 Experimental Setup

Cluster: As infrastructure for our experiments we use GRNET's cloud service, okeanos IaaS[4]. We set up 32 virtual machines as worker nodes. Each worker node provided along with 1 core, 4 GB memory, 20 GB storage and 1 network interface. Also we reserved an additional virtual machine as master node along with 8 cores, 8 GB memory, 20 GB storage and 2 network interfaces (public, private). Each node runs 64-bit Ubuntu Server 12.04.4 LTS.

Systems: Software deployment includes Apache Hadoop 2.5.1, Apache Tez 0.5.0, Apache Hive 0.13.1, Impala 1.4.0 and ADP 2.0. As respect to the Hadoop roles deployment, master node hosts the Resource Manager, the Namenode and the Secondary Namenode, and each worker node hosts a NodeManager and a Datanode. Each Hadoop role configured with 512 MB heap size. Hadoop DFS configured with file block size 128 MB, replication factor 1 and short-circuit reads enabled. Hadoop Yarn configured to use as maximum 3 GB memory per node. For the Tez deployment Application Manager configured with 2.6 GB heap size and intermediate results compression feature enabled.

Datasets: We use the TPC-H benchmark and the Freebase dataset. For TPC-H we generated a total of 0.5 TB of data (or approx. 2.2 billion records) using the generator provided. The benchmark has eight

[1] Apache Hadoop, http://hadoop.apache.org
[2] Apache Tez, http://tez.apache.org
[3] Apache Hive, https://hive.apache.org
[4] GRNET's cloud service, okeanos.grnet.gr

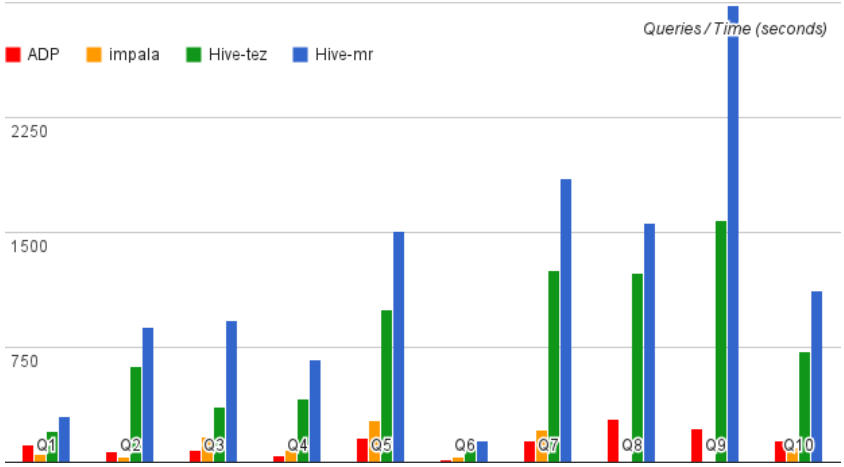

Figure 4.1: Execution time running TPC-H like workload.

tables:

$region(1), partsupp(32, ps\_partkey), orders(32, o\_orderkey)$,
$lineitem(32, l\_orderkey), customer(1, c\_custkey)$,
$part(1, p\_partkey), nation(1)$, and $supplier(1)$.

In parenthesis, we show the number of partitions we created for each table and the key based on which the partitioning was performed. In Hive, we used the *CLUSTERED BY* clause when we created the tables. The replication factor of Hadoop is set to 1 in order to have a clear comparison with our system. In ADP, the tables are horizontally partitioned and distributed to the cluster.

Queries: For our experiments, we use TPC-H like workload. We generate the appropriate datasets with scale factor 64, divided into 32 chunks across the worker nodes. Then, we load the datasets according to the best data placement and format on each system as discussed below. The format of the tables was selected as a tuning configuration on each system. Optimized Row Columnar (ORC) file format provides a highly efficient way to store Hive data. Parquet file format selected respectively to store Impala tables. In Hive queries, the CLUSTERED clause applied on the primary key on each table in order to exploit Bucketed sorted tables. Before running Impala queries COMPUTE STATS clause applied on each table to helps join queries by gathering table and column statistics.

Measurements: We involve the first 10 TPC-H benchmark queries. We run each query as single-user and measure the average execution time of three runs for each system.

4.1.2 Execution Time

In Figure 4.1, we present the results of our experiments based on the above configuration and workload. The most of the systems succeeded throw the workload, except Impala which failed on queries 8,9 bacause the total memory of the cluster was not sufficient. Hive does not appear to be competitive enough compared to Impala and ADP. Regarding Impala and ADP, our system presents consistency throughout the workload and faster execution times on the most of the queries.

Finally, considering our system results compared to the existing hadoop-based SQL-like engines leading us to a new experimental path to exploit hadoop features (high-availability, fault-tolerance) below our system. We have implemented a prototype component which introduce our system in the Hadoop ecosystem. We now study the impact of this component, however the initial results are very promising.

4.2 Just-in-Time Compilation

The engine processes the data in a streaming fashion, performing pipelining when possible, even for UDFs. The UDFs are executed inside the database along with the relational operators in order to push the them as close to the data as possible. This makes possible to process large amounts of data efficiently. Furthermore, it uses tracing just-in-time compilation (JIT) techniques[5] to speedup execution by adapting at runtime to the data. This is equivalent to continuous query optimization and UDF re-ordering based on the input data stream and is very hard to do with conventional methods. JIT also allows the blending of the execution of several UDFs together reducing the number of context switches. The query execution has ACID guarantees as long as the UDFs are side-effect free.

4.2.1 Compare UDFs Interface

In ADP the UDFs are implemented in Python. Both Python and SQLite are not strictly typed, thus enabling the implementation of UDFs with dynamic schemas that can dynamically change based on their input. Algorithm 1 implements the average UDF. The *init* method is called once per group and initializes the internal structures of the operator. The *step* method is called for every record of the group. The *final* method returns the result.

Algorithm 1 Average UDF on ADP

```
 1: class Average:
 2:   def __init__(self):
 3:     self.sum = 0.0
 4:     self.count = 0
 5:
 6:   def step(self, *args):
 7:     self.sum += float(args[0])
 8:     self.count += 1
 9:
10:   def final(self):
11:     yield self.sum / self.count
```

Algorithm 2 shows the implementation of the average UDF on Impala[6]. Comparing the two, we observe that the code for Impala is 6 times larger. Furthermore, notice that ADP performs automatic state serialization and garbage collection so there is not need to implement this functionality. In addition, since python is not strictly typed, we use the same functions for integer, floats, doubles, and even for strings.

Furthermore, it offers a clean syntax making the UDFs first class citizens of the language. An example of the use of the Average UDFs on Shark is as follows:

```
select TRANSFORM(l_discount)
       USING 'python /udfs/avg.py' as (avg_disc)
from lineitem_cached;
```

and in ADP we can express the same query as follows:

```
select Average(l_discount) as avg_disc
from lineitem_cached;
```

4.2.2 Compare UDFs Efficiency

In this set of experiments, we compare the efficiency of the system concerning the UDFs. We used the *Average* function and implemented it as UDF in the systems since is also available as a native function in

[5] http://pypy.org/
[6] https://github.com/cloudera/impala-udf-samples/blob/master/uda-sample.cc

Algorithm 2 Average UDF on Impala

```
 1: struct AvgStruct {
 2:   double sum;
 3:   int64_t count;
 4: };
 5:
 6: void AvgInit(FunctionContext* context,
 7:                 StringVal* val) {
 8:   val->is_null = false;
 9:   val->len = sizeof(AvgStruct);
10:   val->ptr = context->Allocate(val->len);
11:   memset(val->ptr, 0, val->len);
12: }
13:
14: void AvgUpdate(FunctionContext* context,
15:                     const DoubleVal& input,
16:                     StringVal* val) {
17:   if (input.is_null) return;
18:   AvgStruct* avg =
19:     reinterpret_cast<AvgStruct*>(val->ptr);
20:   avg->sum += input.val;
21:   ++avg->count;
22: }
23:
24: void AvgMerge(FunctionContext* context,
25:                   const StringVal& src,
26:                   StringVal* dst) {
27:   if (src.is_null) return;
28:   const AvgStruct* src_avg =
29:     reinterpret_cast<const AvgStruct*>(src.ptr);
30:   AvgStruct* dst_avg =
31:     reinterpret_cast<AvgStruct*>(dst->ptr);
32:   dst_avg->sum += src_avg->sum;
33:   dst_avg->count += src_avg->count;
34: }
35:
36: const StringVal AvgSerialize(
37:     FunctionContext* context, const StringVal& val) {
38:   StringVal result(context, val.len);
39:   memcpy(result.ptr, val.ptr, val.len);
40:   context->Free(val.ptr);
41:   return result;
42: }
43:
44: StringVal AvgFinalize(FunctionContext* context,
45:                           const StringVal& val) {
46:   AvgStruct* avg =
47:     reinterpret_cast<AvgStruct*>(val.ptr);
48:   StringVal result;
49:   if (avg->count == 0) {
50:     result = StringVal::null();
51:   } else {
52:     result = ToStringVal(context,
53:                          avg->sum / avg->count);
54:   }
55:   context->Free(val.ptr);
56:   return result;
57: }
```

SQL. We compare in terms of number of records and number of UDFs in the query. The generic form of

queries we used is as follows:

```
select AVG(X1), AVG(X2), ... AVG(X6)
from (select X1, X2, ... X6
      from lineitem limit <L>);
```

Figure 4.2 shows the running time of the query with one AVG function when varying the size of the table and Figure 4.5 shows the execution time when varying the number of AVG using both the native and the UDF implementation. We observe that ADP is very efficient because it blends the UDFs with the engine itself and is not called as an external function. Furthermore, we observe that our engine is able to scale better when the number of UDFs is higher and we begin the see the effect of the JIT. The effect of JIT appears clearly when the query has a large number of UDFs.

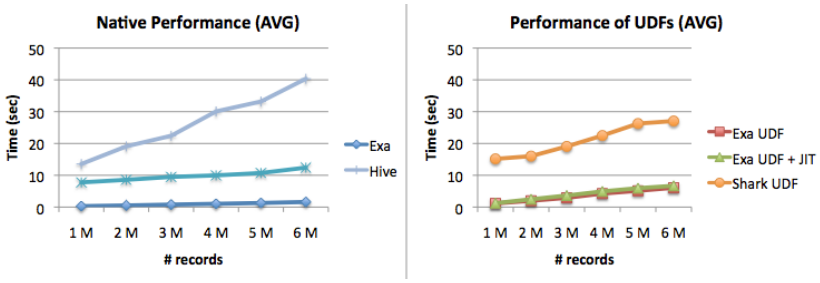

Figure 4.2: Performance varying table size.

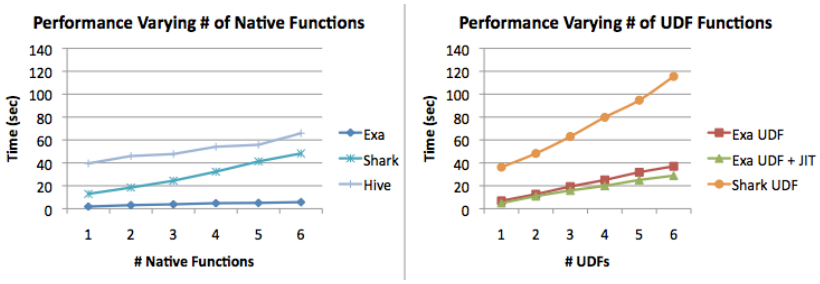

Figure 4.3: Performance varying # of UDFs.

To clearly observe the effect of the JIT techniques, we use the following query that has 9 UDFs:

```
select MEDIAN(d), GMEAN(d), STDEV(d),
       AVG(cd), STDEV(cd)
from (select LEVENDIST(x, y) as d,
             LEVENDIST(shipd, commd) as cd
      from (select REGEXPR("(\w+)[' ']",
                           l_comment) as x,
                   REGEXPR("[' '](\w+)",
                           l_comment) as y,
                   l_shipdate as shipd,
                   l_commitdate as commd
            from lineitem
            limit ...)
where x is not null
  and y is not null
  and d + cd > 4);
```

The Median UDF computes the median of the values, Gmean computes the geometric mean, Stdev computes the standard deviation, Levendist computes the Levenstein distance (or Edit distance) between

2 strings, and Regexpr evaluate a regular expression on the input string (the expression in parenthesis are returned as a result).

Figure 4.4 shows the execution time of the complex query when varying the size of the input. We observe that the tracing JIT techniques are able to blend the UDFs together and are very efficient. Figure 4.5 shows the improvement using JIT of the individual UDFs of the query. We observe that complex UDFs have a lot more opportunities for optimization by the JIT compiler and they are more efficient.

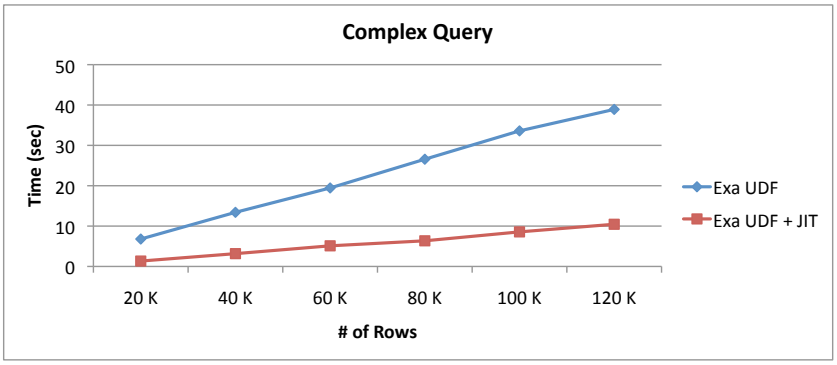

Figure 4.4: Complex query involving 9 UDFs.

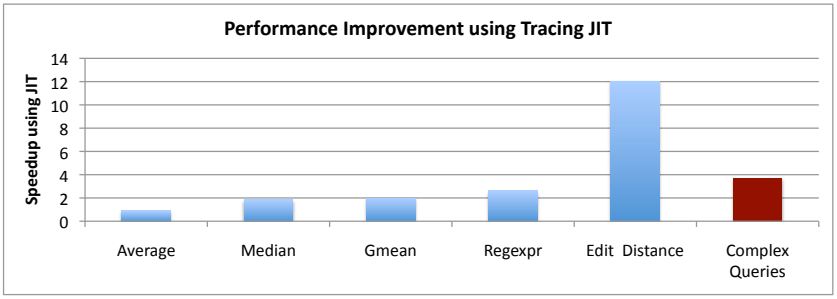

Figure 4.5: Improvement using tracing JIT.

4.3 Compression

Figure 4.6 shows the total time to send 5.5 GB of data over the network using different compression techniques. We observe that our approach is able to perform better and the total time to transfer the data (CPU + network) is the smallest. Furthermore, the compression rate is higher: i.e., the time needed to transfer the data is the smallest. Our algorithms achieve very good performance under high network traffic because their are able to compress the data more efficiently.

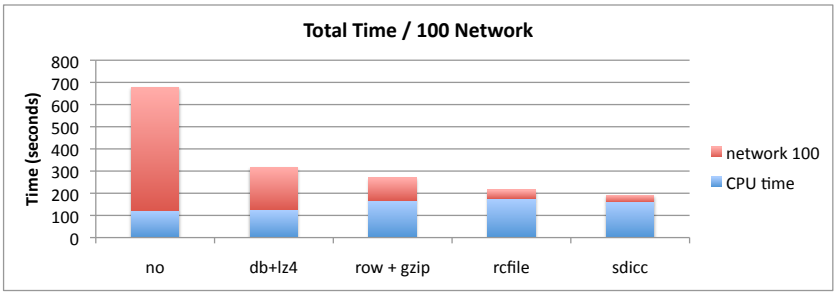

Figure 4.6: Compression algorithms comparison.

Chapter 5

Conclusions

In this deliverable we presented the work that has been done in work package WP7, during the second year of the Optique project. This work can be divided in two main parts. The first one is about temporal queries and stream processing, which has been presented in Chapter 3, whereas the second part is about optimizations and improvements regarding several components of the system that have been presented in Chapter 2. Also, in Chapter 4 we presented experiments that compare the efficiency of our system with other state of the art systems for declarative data processing in cloud environments.

During the third year of the project we will continue working on optimization aspects. First of all we will examine and compare different algorithms for the multi-query optimization problem, also by taking into consideration the particularities of a distributed environment. We will also tackle possible new requirements that will show up regarding federation. Several new aspects of optimization will be considered during this year, like the implementation choice for each operator in the local nodes of the system and also recovery from software or hardware errors. Also, dataflow scheduling and adaptive execution, which have been extensively studied during the first year, will be used and integrated with the other optimization methods.

Another issue that came up as requirement from the use cases, is the execution of geospatial queries. Since ADP uses an extension of SQLite as the engine for local processing, the use of SpatiaLite[1], the spatially enabled version of SQLite, enables ADP to support all spatial functions supported by SQLite and to evaluate them in the database. In the third year we will integrate this solution into ADP.

[1] http://www.gaia-gis.it/gaia-sins/

Bibliography

[1] TPC-H Benchmark, `http://www.tpc.org/tpch/`.

[2] Multi-version concurrency control algorithms. In *Encyclopedia of Database Systems*, page 1870. 2009.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, June 2006.

[4] Diego Calvanese et al. Optique: OBDA solution for big data. In *ESWC (Satellite Events)*, pages 293–295, 2013.

[5] Diego Calvanese, Davide Lanti, Martin Rezk, Mindaugas Slusnys, and Guohui Xiao. A scalable benchmark for OBDA systems: Preliminary report. In *In Proc. of the 3rd Int. Workshop on OWL Reasoner Evaluation (ORE 2014)*, 2014.

[6] Craig Chambers, Ashish Raniwala, et al. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[7] Biswapesh Chattopadhyay et al. Tenzing a SQL implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.

[8] Cloudera. "Cloudera Impala: Open source, interactive SQL for Hadoop, http://www.cloudera.com/".

[9] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In *OSDI*, 2004.

[10] Hector Garcia-Molina et al. The TSIMMIS approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2):117–132, 1997.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[12] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[13] Goetz Graefe and William J McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 209–218. IEEE, 1993.

[14] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, pages 439–450, 2014.

[15] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[16] Alon Y. Halevy, Anand Rajaraman, and Joann J. Ordille. Data integration: The teenage years. In *VLDB*, pages 9–16, 2006.

[17] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[18] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.

[19] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[20] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[21] Andrew Lamb et al. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[22] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[23] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, pages 251–262, 1996.

[24] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11), 2012.

[25] Yucheng Low et al. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[26] Grzegorz Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[27] Sergey Melnik et al. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[28] Konstantinos Morfonios and Yannis E. Ioannidis. Snowflake schema. In *Encyclopedia of Database Systems*, pages 2665–2666. 2009.

[29] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *SIGMOD Conference*, pages 949–960, 2011.

[30] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[31] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Data Engineering, 1988. Proceedings. Fourth International Conference on*, pages 311–319. IEEE, 1988.

[32] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[33] Orestis Polychroniou and Kenneth A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, page 6, 2013.

[34] Prasan Roy, Sridhar Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Record*, 29(2):249–260, 2000.

[35] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[36] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data & Knowledge Engineering*, 12(2):197–222, 1994.

[37] Ashish Thusoo et al. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[38] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[39] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 533–544. ACM, 2007.