LARGE-SCALE ELASTIC ARCHITECTURE FOR DATA AS A SERVICE

Project Number:	FP7-ICT-318809
Project Title:	Large-Scale Elastic Architecture for Data as a Service
Deliverable Number:	D3.2
Title of Deliverable:	LEADS real-time processing platform
Contractual Date of Delivery:	M12 – 2013-09-30
Actual Date of Delivery:	2013-10-2

Abstract

The real-time processing platform is a key component of the LEADS platform, enabling scalable, distributed, real-time processing of massively distributed data across micro-clouds, and supporting efficient query execution for arbitrarily complex queries on a combination of static/historical and streaming data. This deliverable describes the current progress with respect to the platform (the basic design decisions, as well as the abilities of the existing implementation), and gives a brief overview of the work to be performed in the next year, and until the end of the project.

List of Contributors

Name	Organization	E-mail
Xiao Bai	BM-Y!	xbai@yahoo-inc.com
Matthieu Morel	BM-Y!	matthieu@yahoo-inc.com
Ata Turk	BM-Y!	ata@yahoo-inc.com
Emmanuel Bernard	Red Hat	ebernard@redhat.com
Jonathan Halliday	Red Hat	jonathan.halliday@redhat.com
Mark Little	Red Hat	mlittle@redhat.com
Mircea Markus	Red Hat	mmarkus@redhat.com
Manik Surtani	Red Hat	msurtani@redhat.com
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Ekaterini Ioannou	TSI	ioannou@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
Ioakim Perros	TSI	imperros@softnet.tuc.gr
Evagelos Vazeos	TSI	vagvaz@softnet.tuc.gr
Christof Fetzer	TUD	Christof.Fetzer@tu-dresden.de
André Martin	TUD	Andre.Martin@tu-dresden.de
Do Le Quoc	TUD	Do@se.inf.tu-dresden.de
Frezewd Lemma Tena	TUD	Frezewd_Lemma.Tena@mailbox.tu-dresden.de
Lenar Yazdanov	TUD	Lenar.Yazdanov@tu-dresden.de
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch

Document Approval

	Name	Email	Date
Approved by WP Leader	Minos Garofalakis	minos@softnet.tuc.gr	2013-09-30
Approved by GA Member 1	Christof Fetzer	Christof.Fetzer@tu-dresden.de	2013-09-25
Approved by GA Member 2	Xiao Bai	xbai@yahoo-inc.com	2013-09-21

Contents

GLOSSARY

EU	European Union
FP7	Seventh Framework Programme
KVS	Key Value Store
API	Application Programming Interface
REST	Representational State Transfer
URL	Uniform Resource Allocator
SQL	Structured Query Language
ETL	Extract,Transform, Load

1. Executive summary

The real-time processing platform is a key component of the LEADS platform. The core functionality of the platform is to enable scalable, distributed, real-time processing of massively distributed data across micro-clouds, and to support efficient query execution for arbitrarily complex queries on a combination of static/historical and streaming data. This document describes the current progress with respect to the platform (the basic design decisions, as well as the abilities of the existing implementation), and gives a brief overview of the work to be performed in the next year, and until the end of the project.

Clearly, the platform interacts to, and relies on, other LEADS components. First, the platform relies on the Key Value Store (WP2) to retrieve the collected data (i.e., the crawled webpages or the private information explicitly pushed by the user), and to enable communication between processes running across multiple micro-clouds. Second, the platform will interoperate tightly with the Scheduling and Placement component (WP4) in order to optimize the distributed processing tasks and the query execution plans, for reducing energy and increasing performance. Finally, the platform will be a key component for the applications developed for validating LEADS.

The current implementation of the real-time processing platform already supports distributed data processing tasks, even across multiple micro-clouds, based on the location transparency offered by the Key Value Store (KVS). These tasks include indexing of the resources in order to enable efficient answering of user-defined queries (e.g., all webpages containing the term ‘sports’), computing statistics frequently required for query execution (e.g., PageRank), and other tasks specifically defined by the user to satisfy special requirements. The implementation is fully integrated with the up-to-date version of the KVS, for storing the results and for coordinating the data processing tasks across multiple micro-clouds. Furthermore, the KVS listeners mechanism is already employed to trigger distributed – near real-time – processing of all webpages. Currently, only some manually deployed listeners are installed, focusing on answering a set of fairly straightforward pre-defined user queries (in later implementations, appropriate listeners will be created automatically for most types of user queries).

The current platform implementation also offers a command-line interface for executing simple SQL queries (currently, only simple conditional clauses and projections, aggregators, sorting, and equi-joins on a pre-selected set of indexed attributes are supported). Finally, to satisfy the requirement of the end-users for an importance score for webpages, the platform includes a novel streaming PageRank algorithm developed specifically for the LEADS infrastructure, which enables continuous maintenance of the PageRank scores of all crawled webpages. The algorithm is fully integrated with the KVS, receiving and processing all crawler updates as a distributed stream.

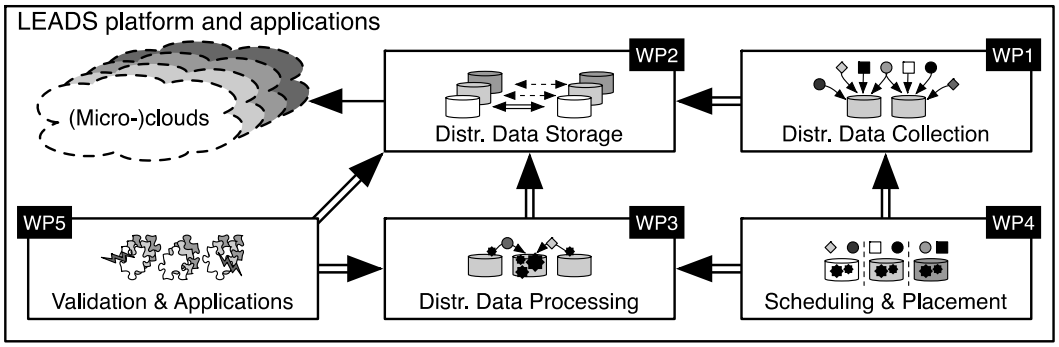

Figure 1. The real-time processing platform within LEADS

2. Introduction

The purpose of this document is to summarize the current status of the LEADS real-time processing platform, explain the main design decisions, and detail on the ongoing and future work. The document is structured as follows. In the next section we present a brief overview of the real-time processing platform requirements and architecture. In Section 4, we discuss the query processor engine, presenting the current status – functionality, algorithms, and implementation details – as well as future plans. Section 5 describes our progress on cross-cloud real-time processing, and in particular on streaming PageRank maintenance, presenting a distributed algorithm that is specifically made for the multi-cloud LEADS architecture. In Section 6 we present a brief overview of other topics in the context of the real-time processing platform, on which we have some progress, but still do not have mature results integrated in the project. Section 7 provides instructions on downloading and running the prototype implementation, and Section 8 concludes the document.

3. Brief overview of the work-package requirements and architecture

The real-time processing platform is a key component of LEADS, responsible for the scalable processing of a combination of massive historical data and streams. The main novelty of the platform – and what sets it apart from earlier platforms for distributed stream processing, like StreamMine and MapReduce Online – is that it will be distributed across multiple micro-clouds, which might even reside at different distant networks. The user queries will be analyzed, and distributed into the participating micro-clouds, which will monitor the corresponding web streams, and generate the query results. The distribution of the queries to the participating clouds, as well as the generation of the query execution plans, will be taking into account different cost factors, like energy consumption, required network resources, computational cost, and execution time.

The main functional requirements of the real-time processing platform are as follows:

1. Scalable and real-time processing of massively distributed dynamic data over multiple heterogeneous micro-clouds;
2. Novel programming models and tools to ease the development of distributed data mining algorithms, and intuitive querying interfaces to provide basic data analysis and mining functionalities;
3. Methods to optimize the performance and energy efficiency of distributed data analysis and mining in LEADS;
4. Privacy-preserving query processing

The platform will be able to offer robustness and fault tolerance for the above functionalities.

To perform these functionalities, the platform will rely on components constructed by other LEADS work-packages. Precisely, it will rely on the **consistent distributed data storage** component, offered by WP2, in order to combine processing of stored and streaming data, but also to store the query results and necessary meta-data. Furthermore, it will employ the **scheduling and data placement functionality** of WP4 in order to decide on the optimal query execution plan and distribution of processing tasks to micro-clouds. Figure 1 summarizes these interactions. The APIs for the two components of WP2 and WP4 are presented in the corresponding deliverables D 2.2 and D 4.2.

3.1 Main components of the real-time processing platform

Three main components comprise the real-time processing platform: (a) the real-time processor, described in the next Section, (b) the listeners attached to the KVS, discussed at Section 3.1.2 and (c) the query processor, presented in Section 3.1.3.

3.1.1 The real-time processor

This component is responsible for the real-time processing of all data accessible by LEADS, i.e., the crawled webpages, and any data inserted by the user via a push interface. The main architecture of the component is described in Figure 2. The component will scale across multiple micro-clouds, achieving load balancing and scalability. An extensible architecture has been chosen to materialize this functionality, which will enable users to attach ad-hoc components for satisfying particular requirements. Such components may handle, e.g., text extraction from different document formats, basic stemming operations, and code for extracting and maintaining different statistics and query-specific historical data (for example, for maintaining an inverted index). Finally, since PageRank is currently the main ranking technique for query execution purposes in the Web (clearly, in combination with query relevance measures), a key functionality of the real-time processor is a streaming PageRank maintenance for the crawled web-graph. A novel LEADS-based PageRank algorithm is described in Section 0.

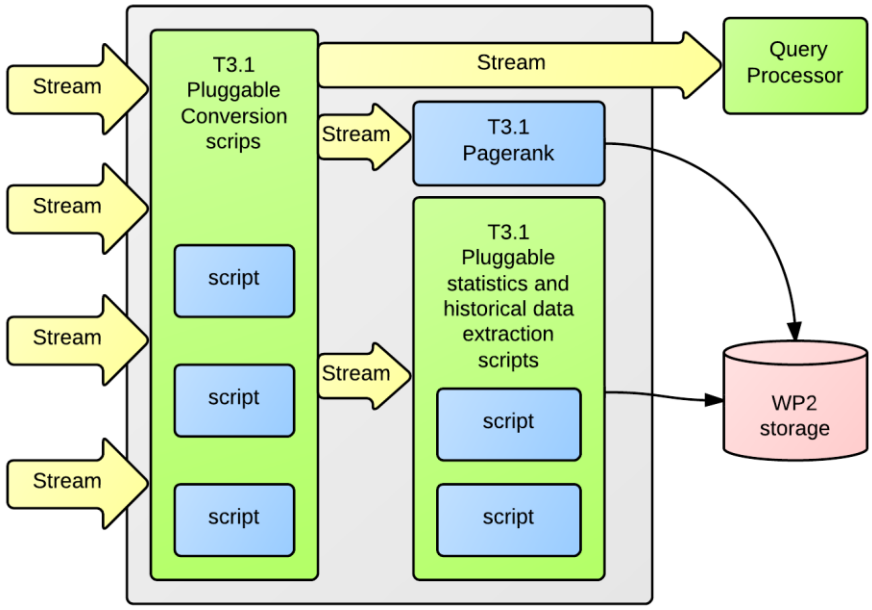

Figure 2. The real-time processor

3.1.2 Listeners attached to the KVS

In order to feed the real-time processor, we will be attaching listeners at the KVS, which monitor all updates, i.e., all puts. Whenever an update satisfies a listener's condition (e.g., it inserts a newly-

crawled webpage), it will be forwarded to the appropriate node query executor for further processing. The listener's conditions will be determined by the running queries, and installed at the KVS using a dedicated API. For instance, in order to keep the average sentiment value over all webpages in the .com domain, a listener will be installed to check the URL of all newly-crawled webpages, and forward to the appropriate node query executor only the updates belonging to the domain for further processing. Notice that, to avoid bottleneck and scalability issues at the KVS, only inexpensive filtering functionality will be pushed to the listeners. The expensive computation, such as sentiment analysis, will instead be pushed to the query processors, and the real-time processors.

3.1.3 The query processor

The query processor focuses on efficiently executing user queries. The coarse-grained architecture of the component is presented in Figure 3. The main sub-components are: (a) the query description interface, which enables users to define their information needs in a flexible and straight-forward approach, (b) the query planner, which derives efficient query execution plans across multiple micro-clouds, (c) the deployer, which deploys, coordinates, and monitors the query execution at the micro-clouds, and, finally, (d) the node query executor, which runs at each node in the network and handles the actual query execution. In the following section, we will be discussing query processor's components in more details.

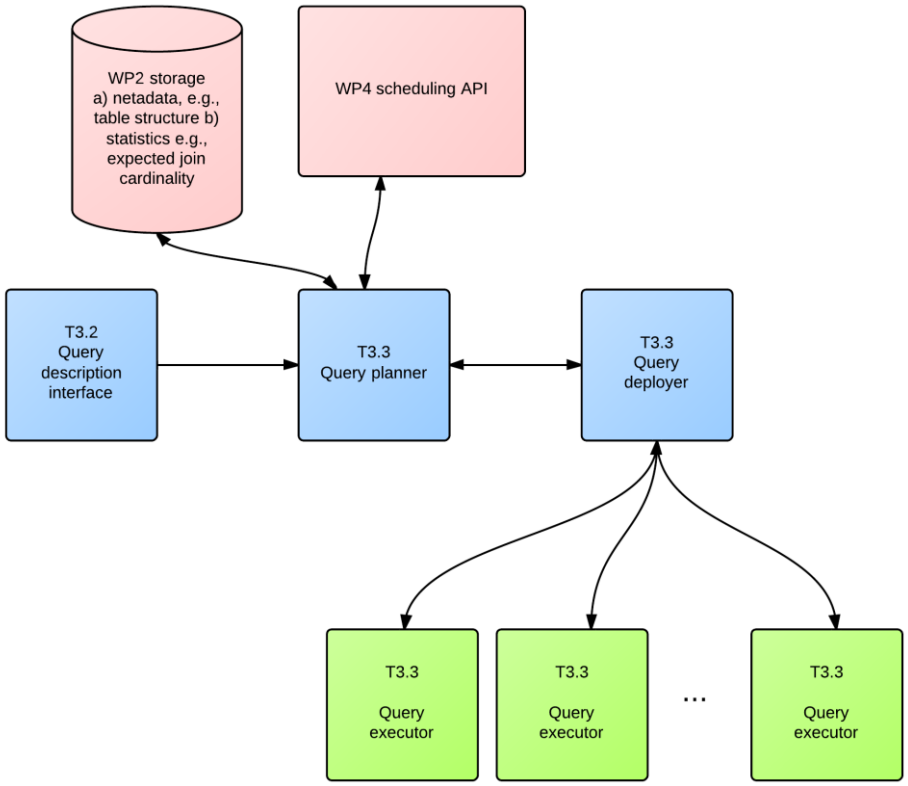

Figure 3. The query processor

4. Query Processor Engine

In this section we elaborate on the architecture of the query processor, providing a detailed description of its design and functionality.

4.1 Query Description Interface

The Query Description Interface (Figure 5) consists of two modules for describing queries, a graphical tool and an interface accepting queries described with a declarative language. Precisely, novice users are supported by a simple and intuitive graphical tool based on mashups. The tool enables combination of simple widgets (the so called connectors), each one executing a basic functionality, such as retrieving data from the KVS, filtering data based on a clause, performing a join or deploying pre-defined MapReduce jobs. Users may combine connectors to form arbitrarily complex combinations, called query workflows. An example query workflow involving a join over two KVS filters and sorting of the final results is presented in Figure 4. This graphical tool is based on Apatar[1], an open-source data integration tool, and is described in more detail in Section 0. On the other hand, expert users rely on a simple interface to express queries in a declarative language. This offers a higher flexibility to the users for describing their query requirements, possibly even including specific optimizations.

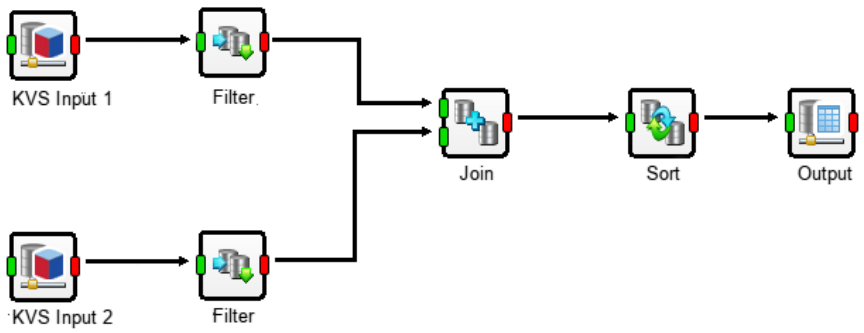

Figure 4. Sample Query Workflow

To decouple the query management from the user interface implementation and to allow an API-based interaction with the LEADS platform, the Query Description Interface component includes the User Interface Manager module, which handles the interaction of the user interfaces with the LEADS platform. The module exposes the following REST API:

a) **processQuery(***String query***):** This function is used for submitting a query written in the declarative language.
b) **processQueryWorkflow**(*Workflow workflow***):** Used for submitting a query workflow
c) **isCompleted(***QueryId Id***):** Returns true if the query is completed and false otherwise
d) **fetchResults(***QueryId Id***):** Used for retrieving the query results from our system.

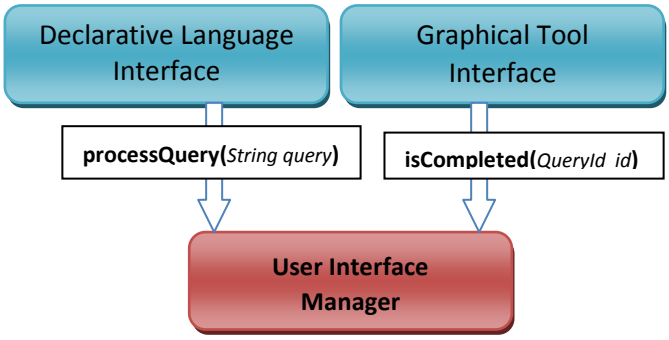

Figure 5. Query Description Interface Design

[1] http://www.apatar.com

When User Interface Manager receives a new query, it replies with the new query's identifier. Following, it transforms the query into an internal query representation and sends it through a Java Message Service queue to the Query Planner component.

The current implementation of Query Description Interface has a functional Declarative Language Interface, which relies on an open-source library (JSQLParser) for parsing simple SQL queries. With respect to the Graphical Tool Interface, we have already designed LEADS-specific Apatar connectors, and we are in the progress of developing them and integrating them in the LEADS platform. In the coming year, we plan to develop a more suitable declarative language interface than SQL, or extend an existing one, such as the Continuous Query Language or StreamSQL, in order to satisfy LEADS requirements for combining streaming and static data in the queries.

4.2 Query Planner

The Query Planner is responsible for extracting efficient execution plans for the queries (analogous to the query planning in relational and distributed databases), which dictate how the actual data processing will be executed. Briefly, query plans extracted by this component describe the operators to be executed, and their execution order and dependencies. For example, consider the following query, expressed in SQL:

```
SELECT url, body, PageRank FROM WebPages WHERE
body contains "ADIDAS" ORDER BY PageRank LIMIT 100
```

With this query, the user wants to find the top-100 webpages that contain the term "ADIDAS" and the result to be sorted by webpages' PageRank.

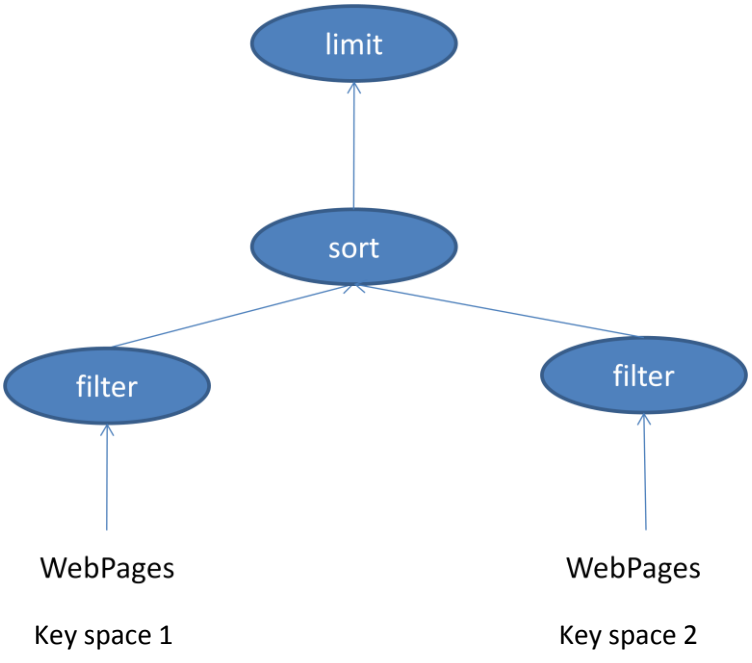

Figure 6. A Sample Query Execution Plan

A sample query plan for the aforementioned query (not necessarily optimal) is depicted in Figure 6. A Sample Query Execution Plan**.** The plan assumes that there are only two key spaces in the KVS, which contain the crawled webpages. Thus, Query Planner creates two identical chains, one for each key space, which filter the webpages that contain term "ADIDAS". Finally, the sort operator merges and sorts the result, and the limit operator discards the redundant webpages and keeps only the top-100.

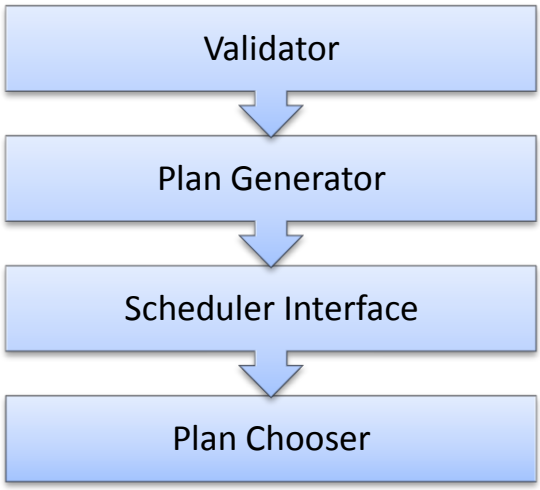

Figure 7. Internal Representation of Query Planner

Figure 7 shows the four modules of the Query Planner component:

a) Query Validator
b) Plan Generator
c) Scheduler Interface
d) Plan Chooser

Query Planner first validates each new query, and breaks it to basic operators, which can be implemented as MapReduce jobs. These operators are passed to the plan generator, which is responsible for generating a set of candidate and equivalent query execution plans. For the generation of efficient plans, the plan generator exploits a set of statistics stored at the KVS, which can be either offered directly by the KVS (e.g., the total number of keys stored in the KVS), or generated by deploying dedicated KVS listeners (e.g., the total number of crawled webpages, or the number of webpages containing a particular keyword). Following, the planner sends these candidate plans to the scheduler (WP4), in order to estimate the execution cost of each plan. In order to help the scheduler make this decision, each operator in a plan is annotated with its network and processing requirements. The resulting query plans, accompanied with their estimated execution cost are returned back to the plan chooser, where the plan with the minimum estimated cost (possibly, according to a complex cost aggregate function) is selected, and passed to the Query Deployer component.

A simplified version of the Query Planner is currently implemented, which generates a single plan, and supports basic operators such as simple conditional clauses, projections, aggregators, sorting, and equi-joins on a pre-selected set of indexed attributes without using any advance optimization techniques for plan generation. Moreover, the scheduler interface is not yet integrated into our system. During the next year of the project we plan to develop optimization techniques to minimize the data transfers inside and across micro-clouds, and study the process and problems of combining streaming and static data over the LEADS infrastructure.

4.3 Query Deployer

The Query Deployer is responsible for deploying and monitoring the execution of a query plan, as well as its recovery in case of either a node or a complete micro-cloud failure. In order to implement the required functionality, the Query Deployer is split into three modules as shown in Figure 8:

a) The **Monitor module**, used for monitoring the execution of the current running plans, and for detecting failures.
b) The **Deploy module**, responsible for installing the appropriate listeners to the KVS for continuous queries, and for deploying the plan's operators to the Query Executor components running at the nodes.
c) The **Recovery module**, which is activated when a failure has been detected. The recovery module attempts to resume the execution of the plan without restarting the whole plan's execution.

When the Query Deployer component receives a plan for deployment, the deploy module is notified to perform the required initialization, i.e., to deploy the listeners to the KVS and the plan operators to the distributed Node Query Executors. The Monitor module keeps track of the execution progress, by monitoring the progress of each operator assigned to the Node Query Executors. Upon completion of any operator, Node Query Executors notify the Monitor, which in turn notifies the Deploy module to start the next operator in the plan. Furthermore, whenever the Monitor module detects a failure during the execution of a plan, it notifies the Recovery module, which activates the recovery procedure for the plan.

In our current implementation, the Query Deployer contains only the Deploy and Monitor module. The Monitor module can monitor the execution of the plan and detect failures, but no recovery action is taken yet. Automatic deployment of the listeners is also not yet fully supported; for the supported queries, all required listeners are manually installed in the KVS. Finally, there is no integration yet with the elasticity mechanism offered by the interface of WP4 (for starting/stopping virtual machines).

Figure 8. The Query Deployer

4.4 Node Query Executor

The Node Query Executor is the core processing unit of the platform, handling all requests for executing operators. These operators can be classified into two types:

a) Traditional operators, such as SELECT, PROJECT, and GROUP BY.
b) Continuous/streaming operators, which are used to materialize stream algorithms,

The streaming operators are focused on event stream processing. The most frequent source of event streams is by the KVS listeners. In particular, listeners are triggered by updates in the KVS. Whenever a listener's condition is satisfied, then the listener generates an event, which is delegated to the appropriate streaming operators. The listener's conditions are determined by the running queries, and installed at the KVS using the dedicated API.

Each Node Query Executor has a module responsible for maintaining and updating the statistics written in the KVS and used by Query Planner's generator, e.g., the selectivity of a join, or the number of URLs. After the completion of an operator, the Query Executor component reports the completion to the Monitor module of the Query Deployer, and saves the results of the operator in the KVS.

The current implementation of the Node Query Executor can handle a partial set of operators, i.e., simple conditional clauses and projections, aggregators, sorting, and equi-joins on a pre-selected set of indexed attributes. The implementation is based on the MapReduce capabilities provided by Infinispan. In the following year, the implementation will be extended to handle a full set of SQL operators. Furthermore, we will investigate the problem of fault-tolerance for the node query executor: currently, a failure at a node running the query executor may lead to failure on executing the query, or to incomplete answers, depending on the operator assigned at the node and the type of failure. Moreover, we will develop a toolbox offering generic data mining tasks, e.g., finding all stream items that satisfy a pattern chosen by the user or finding the top-k frequent keys observed in a stream. Finally, we will work towards enabling users to use a combination of streaming and traditional operators in their queries.

Figure 9 summarizes the steps necessary for processing a query. With green we have marked the components and steps that are currently integrated in the platform (yet mostly in preliminary versions, without any optimization) and with grey we marked the components that are still being implemented.

Query Processing Steps:

1. Design query with the available user interfaces
2. Use the User Interface Manager to submit query to our system
3. The User Interface Manager responds with the query identifier for that query
4. The User Interface Manager transforms the query to internal representation and sends it to Query Planer
5. Query Planner's validator validates the query and breaks it to basic operators
6. Using the statistics in the KVS, the Plan Generator creates a set of candidate plans
7. Using Scheduler's API each query plan is evaluated, to estimate the execution cost
8. One plan from the set is chosen
9. Send the chosen plan to the Query Deployer
10. The Deploy module starts and installs all the necessary listeners for the query
11. The Monitor module is informed from the Deploy module for the deployment of operators
12. Operators are deployed to the Node Query Executor components
13. The static operators implemented in the Node Query Executor read data from the KVS
14. Query Executor reports the completion to the Monitor module in order to continue the execution of the plan
15. The streaming operators handle the stream of events generated by the listeners on KVS
16. After finishing the execution of the whole plan, the Query Deployer changes the status of the query to completed status, so user interfaces can fetch the results using the User Interface Manager API

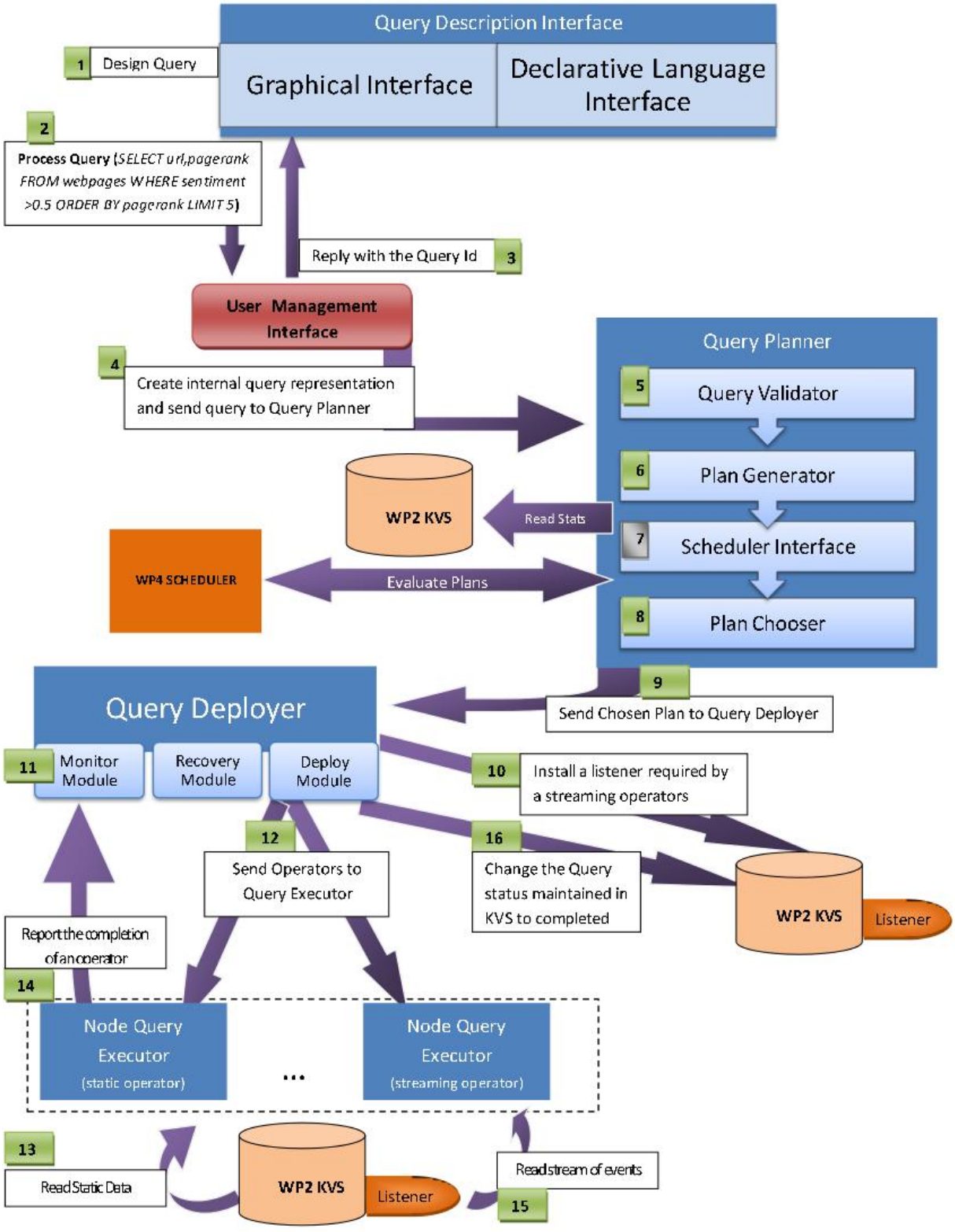

Figure 9. Detailed diagram of the query processing flow in our system

4.5 Future directions

A functional version of the query processor engine is provided with this prototype. The engine directly inherits the scalability properties of the KVS and Infinispan's MapReduce, i.e., it can scale by just adding more nodes in the network, in terms of both computational capacity and storage capacity/memory.

However, as already anticipated for the first prototype in the DoW, the M12 prototype is subject to a number of limitations. The three main limitations, which we will be considering in the following year, are the following. First, the prototype is not yet robust to network and hardware failures. In fact, an unexpected termination at a node may lead to information loss, since the KVS segment running on the machine will not be given the chance to terminate gracefully, e.g., move the data to another node. A hardware or network failure will also affect any listener running on the node, thereby causing loss of information. Clearly, these problems are common to all large-scale data-intensive systems, where network and hardware failures are frequent, and there exist substantial research results in the literature (see, e.g., Storm [Sto13], S4 [NRN+10], and StreamMine [Fet10]), which we will be considering in the next year in collaboration with WP2 and WP4.

Second, the prototype does not yet include all envisioned optimizations. Clearly, integrating the WP3 query engine with the scheduler (WP4) will enable the extraction and materialization of more efficient query execution plans, which take into account the hardware resources and the cost factors of the participating clouds. Furthermore, network resources can be substantially reduced by incorporating synopses/sketches [BB70, CM04, PGD12] and efficient distributed monitoring algorithms, e.g., the geometric method [SSK06]. Finally, a statistical solution for automatically extracting a set of listeners that will reduce the computational cost will be considered (e.g., for maintaining secondary indexes over the KVS).

Finally, the query processor engine functionality will be enriched in the following year in order to efficiently support a more advanced SQL syntax, as well as continuous queries. This will involve investigation on alternative query description languages (such as extensions of CQL [ABW06], or mashup-based user interfaces [Apa13]), as well as research on distributed query monitoring.

Notice that all described limitations are non-trivial, stem from open research challenges, and are commonly found in most distributed data-intensive systems. Therefore, we anticipate that substantial research resources will be invested on investigating and addressing these limitations, and on integrating the results in the LEADS platform.

5. The multiple micro-cloud PageRank maintenance algorithm

To adapt to the requirements of Web users, our framework should have the ability to rank webpages, not only based on the relevance score with a query, but also based on the 'importance' of each webpage. In fact, due to the explosion of the size of the web in the last decade, ranking algorithms have become one of the most important components of modern search engines, such that users can quickly focus on the most important relevant results for their query out of the, possibly thousand, relevant web-pages. PageRank is one of the key algorithms for performing this ranking, empowering Google web search engine.

Similar ranking functionality is also required at the LEADS project, as evident by discussion with the end-user, during the validation scenario development. Our solution is custom-designed for LEADS, as it inherently considers: (a) the distribution of the processing units to (possibly distant) micro-clouds,

and, (b) the continuous maintenance of the webpages scores according to the streaming/continuous updates. No existing PageRank computation algorithm satisfies both of the above requirements.

5.1 Preliminaries – Related Work

Page et al. [PLB+99] remarked that the importance ranking of a URL is essentially the determination of the limiting distribution of a random walk on the web graph. This coupling of the theory of random walks with the PageRank computation, gave rise to a family of Monte Carlo algorithms for PageRank computation. The core concept behind all these algorithms is to execute random walks on the available web-graph, counting the visit counts at each webpage. The final number of visit counts per webpage is in fact proportional to the PageRank of the page. Notice that these methods can be distributed by nature (yet under the shared memory model) and can be very efficient in dealing with our streaming query needs, as they find suitable application to settings where fast and approximate answers are preferable than slower and exact ones. Fundamental work based on this family of methods for static web graphs is provided by Avrachenkov et al. [ALN+07].

To enable continuous maintenance of the PageRank vector, Bahmani et al. [BCG10] recently proposed the state of the art PageRank computation algorithm for streaming graphs. They consider a distributed shared memory cluster as the storage medium. Intuitively, this method emulates the standard Reservoir Sampling technique ([V85]), a randomized algorithm for choosing k samples from a list containing n items, where n is either a very large or an unknown number. In essence, the algorithm in [BCG10] attempts to maintain the same visit counts across webpages that would have been the result of the algorithm in [ALN+07], if the whole network was available beforehand, in a static manner. In this way, the PageRank values of all pages can be progressively maintained.

5.2 High-level description of the LEADS-oriented approach

Focusing on networks of remote micro-clouds, such as the ones enabled by LEADS, we have developed a novel PageRank algorithm that enables both distributed PageRank computation, as well as incremental maintenance, i.e., maintaining PageRank with updates on the web-graph. Our algorithm belongs to the Monte Carlo family, constructing random walks and relying on the visit counts at each webpage to estimate its PageRank score. However, unlike the algorithm of Bahmani et al. [BCG10] that enables updates, our approach does not require an explicit maintenance of the full random walk segments. This property enables the participating nodes to exchange only aggregate information whilst creating the random walks, drastically cutting down the network requirements. The algorithm makes no assumption about the assignment of pages to micro-clouds or nodes, e.g., hash partitioning or any other partitioning determined by the crawling policy can be used.

During the initialization, *R* random walks are constructed from each node, each of which terminates at each step with a predefined stopping probability. Since the exact random walks need not be maintained (only the visit counts are important for the algorithm), nodes exchange only aggregate information for constructing the random walks. For example, consider pages A and B, with A linking to B. Whenever the node holding page A discovers $k \geq 1$ random walks that need to proceed from A to B, it only needs to send a single message $< B, k >$, to the node holding B. Compared to the algorithm in [BCG10], which sends k different messages along with the different random walk ids and positions at these random walk segments, this aggregation enables substantial network savings. To emulate the deletion process of random walks in the absence of the already constructed segments, we introduce the notion of *negative* random walks. Negative random walks are constructed identically to standard random walks, but decrease the visit count of the pages passing through,

instead of increasing it. Clearly, the challenge is to show that the distribution of visit counts produced by this method is also proportional to the PageRank vector, as in the case of [BCG10].

A working prototype of the above high-level description is already integrated in the LEADS platform; we are still designing and implementing many algorithmic optimizations that have the potential to substantially improve the performance of the algorithm. Further details are omitted due to the unpublished status of this work.

6. Brief progress report on other topics of WP3

In this section we briefly comment on functionality of the real-time processing platform, for which the work-package partners already started working on, but still do not have solid results integrated in LEADS. These are: (a) privacy-preserving query processing for range and point queries, (b) querying tools, and, (c) results authentication.

6.1 Privacy-preserving query processing

To enable LEADS users to securely use the platform for processing and storing also their own sensitive data, WP3 partners already started looking into privacy-preserving query processing (officially starting at month 12). Our research currently focuses on addressing range queries on encrypted fields, i.e., queries with range conditions. To illustrate the problem, consider the simple data presented in **Error! Reference source not found.**, which could be an excerpt of a national tax registry database. Assume that the salary of each citizen constitutes strictly sensitive information, which should never be made available to unauthorized third-parties. However, due to the magnitude of the dataset, and to enable truly-distributed and scalable query execution, both these fields also needed to be stored in the platform, which may incorporate third-party (untrusted) micro-clouds.

Even though there exist many different encryption techniques, which allow safely storing sensitive data in the cloud, these techniques naturally impose some functionality constraints. Currently, we are looking into the problem of executing range queries on these encrypted data. Two such queries may be (expressed in SQL):

```
Q1: SELECT * FROM citizens WHERE salary≥1K AND salary≤2K
      and
Q2: SELECT COUNT(*) FROM citizens WHERE salary≥1K AND salary≤2K
```

Both queries need to perform a selection on an encrypted field (salary). Clearly, the user does not want to release the decryption method (e.g., a secret key) to the whole platform, since the platform may include compromised computers. The trivial solution of retrieving all records at a trusted computer and decrypting them is also problematic, since it does not scale. Therefore, we need a method that enables distant micro-clouds that hold the information to decide whether the salary is within the query range, without actually decrypting it, and without revealing any information.

TupleID	Salary
T1	1K
T2	2K
T3	2K
T4	2K
T5	5K
T6	5K
T7	6K
T8	8K
T9	8K

Original Data

Figure 10. An excerpt of a national tax registry database (id,salary)

Related work comes from two distinct areas: (a) data confidentiality, and (b) access privacy. In the following we describe the key representatives from these areas.

Data confidentiality

Order Preserving Encryption

In the recent years significant research is conducted around the subject of Order Preserving Encryption Schemes (OPE) [BCY09] [XYH12] [YKK12] [LW12] [PLZ13]. OPE schemes are deterministic encryption schemes that preserve the order of the plaintext, allowing the execution of efficient range queries directly on encrypted data on the cloud. An ideal security guarantee for OPE schemes has to ensure that no other information is revealed besides the order of the plaintext values; Among the family of OPE schemes, only the recent work of Popa et al. in [PLZ13] achieves the ideal OPE security guarantee. However, all OPE schemes including [PLZ13], have two major drawbacks which are the order revealing and the distribution leakage of repeated ciphertexts due to determinism. Hence, an adversary who is either aware of the domain of the encrypted values or gathers statistical data, is capable of building a mapping between the actual and the encrypted values.

Bucketization Approaches

Hacigumus et al [HILM02], proposed a bucketization based approach that Hore et al. [HMT04] further extended and improved. The bucketization technique presupposes data partitioning into buckets that are eventually stored in the cloud. The client side retains the respective indices whose number increases linearly with the number of buckets, also affecting the index search process in the same manner. Whenever a range query is to be executed, all buckets containing query results are retrieved from the server. However, false positive values are also included in the retrieved buckets and need to be filtered out from the final answer. This process is carried out on the client side and requires the decryption of all tuples contained in the buckets leading to an often prohibitive cost. Furthermore, updating data is expensive since it requires re-distribution of the tuples. In terms of security this approach is less threatening to the Access Privacy Problem due to false positives, but does not provide any proofs. More specifically it tunes security to the desired level with the cost of degrading efficiency respectively (uses the trade-off between efficiency and privacy).

Predicate Encryption

Predicate encryption query schemes guarantee strong security definitions, while causing at the same time high computational overhead since they attempt to allow privacy preserving querying by multiple authorized users. In predicate encryption schemes a ciphertext is associated with a set of hidden attributes S. We define a predicate function f() and a token created by the master secret key. Using this token in function f() the client can check whether this secret set for a ciphertext is satisfied

(f(s)=1) without decrypting the data. Boneh and Water support range queries in [BW07], since they achieve strong security guarantees. However, their approach suffers from big ciphertext size. Shi et al. in [SBC07] and Lu in [Lu12], attempt a relaxation on Boneh's encryption scheme on the security guarantees in order to improve the efficiency. The aforementioned approaches guarantee Data Confidentiality, but they do not provide any Access Privacy guarantees.

FHE - HE

Homomorphic cryptosystems(HE) cannot support privacy preserving range queries, since an appropriate range query homomorphic cryptosystem does not exist. A fully homomorphic encryption (FHE) scheme, proposed by Gentry in [Ge09] and in [Ge11], is a generalization of homomorphic encryptions that allows the execution of any arbitrary function, including range queries on encrypted ciphertexts, without requiring any decryption. Yet, it appears to be totally impractical due to the required ciphertext size and the computational time that sharply increases as we increase the security level.

Access Privacy

Querying on data may lead to the reveal of sensitive information about the data and in such cases access privacy is essential. Among the proposed protocols guaranteeing Access Privacy, the most prominent ones are Private Information Retrieval [CKG98] [SiC07] [OG12] and Oblivious RAM [Ostr90] [WS08] from the Crypto community. It is worth noticing that these approaches are potential solutions to the aforementioned problem, but they introduce an additional computational overhead which in practice is prohibitive.

Our direction for solving this problem utilizes techniques from the Database community. Our aim is to simultaneously preserve Access Privacy and Data confidentiality, in an efficient manner. To the best of our knowledge none of the prior works has achieved the aforementioned integration. We refrain from including the details of the approach in this document, since the work is still unpublished. For further details, please contact us.

6.2 Querying tools

The current implementation supports only SQL queries via a command-line terminal. However, this will not be a constraint of the final LEADS platform. The WP3 members have started extending Apatar such that it can be used as a querying interface over LEADS for inexperienced users. Apatar is an open source data integration and ETL tool, and relies on connectors to perform data loading, integrations and transformations (e.g., to retrieve data from a database, or to perform basic text processing and transformation). Constructing an ETL process is as simple as selecting the proper connectors from a prepared library, parameterizing them, and linking them to a mashup.

The existing connectors library of Apatar does not completely cover the LEADS requirements. In particular, up to now we have identified and started constructing the following connectors:

a) A connector to enable exchanging data with the KVS
b) A connector to enable deploying and monitoring map-reduce jobs
c) Connectors to support frequent processing on streams, as well as arbitrary stream processing tasks
d) A set of connectors for constructing basic SQL queries (equi-joins, group by and order by clauses), on both streaming and static data.

In the course of the project, we plan to fully implement these connectors, connect them with the platform, and possibly identify additional connectors that implement frequent end-user requirements.

During the second year of the project, we will also focus on developing/extending a declarative language that will enable expert users to perform complex queries. Declarative programming, compared the imperative paradigm (e.g., MapReduce, or standard programming languages), provides better flexibility for optimizations in distributed settings (see, e.g., Bloom [ACC+10, ACH+11], or OverLog [LCM+06]).

6.3 Results authentication

Our previous discussion was based on an inherent assumption that the servers are curious to learn/decrypt sensitive data, but not malicious. However, outsourcing naturally raises the issue of *trust*. Specifically, the third party may act maliciously to increase profit, e.g., it may collude with rival companies and present fraudulent results to bias the competition; or it may shed some of the workload and only compute on a sample of the input to save effort. Even when the server is honest, problems can arise, as it may run buggy software, or (given the scale of the problems considered) suffer from equipment failure or read/write errors.

It is therefore particularly important to adopt methods for *result authentication*. In a recent work [PCDG13], we have proposed such methods with a particular focus on continuous queries. These methods enable the clients to verify the correctness of the streaming results they receive from the server, i.e., that they have not been tampered (*integrity*) and up-to-date (*freshness*). The goal of the work is to make stream authentication a very lightweight operation for all parties involved, and establish it as a standard tool for error-checking, in a similar way to the ubiquitous use of checksums for reliable file transfer. Briefly, the main contributions of the work are:

- We introduce constructions for authenticating: (i) *sums of dynamic vectors* produced by one or multiple streams, (ii) *dot products* of dynamic vectors produced by different streams, and (iii) *products* between *dynamic matrices* generated by different streams. Our schemes are extremely lightweight for the owner, as they mainly involve inexpensive hash operations and modular additions or multiplications in a very small finite field. They are also cheap for the client, who verifies the result without adding substantially to the cost of reading the output. Moreover, they impose only a small extra overhead to the computation cost of the server.
- We provide strong *cryptographic* guarantees for all our constructions, derived from formal definitions and proofs.
- We show how to adapt the basic schemes in order to solve a range of database queries in stream authentication, including group by queries, joins, in-network aggregation, similarity matching, and event processing. To our knowledge, we are the first to address result authentication for such a large range of complex queries.

The generic idea is to design proper summaries, which are different for each class of queries that the owner(s) of data can efficiently maintain upon the arrival of new data. At each time period, the owner forwards to the server not only the newly arrived data, but also a compact signature, which is generated by the owner's maintained summaries. The server is also provided with a way to combine these signatures, in order to provide a proper proof to the client(s) that its reported results are current and have not been tampered. The client can then easily verify the integrity and freshness of the provided results. More details for the work can be found in [PCDG13] (available in the appendix).

Our current focus is on combining the proposed methods with encryption, to be able to support result verification also on encrypted data. Furthermore, we will explore different query types, such as arbitrary sliding window queries and optimizations involving cases of large numbers of clients.

7. Prototype implementation

The source code can be found online at

`https://github.com/vagvaz/Leads-QueryProcessor.git`

Furthermore, the prototype implementation is deployed at

`https://dashboard-dresden4.aocloud.eu/project/images_and_snapshots/`

as a snapshot of a virtual machine with name

`Leads-QueryProcessor Demo`

The virtual machine image includes the following software:

- An open-source operating system (Ubuntu linux), including the Java Virtual Machine (OpenJDK 1.7.0).
- The Query Processor Engine of WP3
- The distributed PageRank algorithm of WP3
- The terminal-based user interface for executing SQL queries of WP3
- A deployment of the Key-Value Store of WP2
- The distributed web crawler of WP1

The snapshot runs at IP address `80.156.223.205`.

Notice that additional snapshots can be started in order to increase scalability (no configuration is required, as long as the virtual machines run in the same subnet).

7.1 Running the prototype

To run the prototype, first create a file containing the private key provided for the demo. Then, in order to connect to the already-running virtual machine, use the following command:

`ssh -i path_to_file ubuntu@80.156.223.205`

After successful connection to the virtual machine, the LEADS query processor and user interface can be started using one of the following scripts:

processor-with-crawler.sh: This is a stand-alone script, which starts an instance of a web crawler together with the query processor. This script is provided only for demonstration purposes and will not be useful in the final installation, since the crawlers and the query processors will be running independently, possibly also in different micro-clouds.

processor.sh: This script starts an instance of the query processor. Use this script if there already exists a running instance of the web crawler in the same subnet (see D1.2 for more details on the web crawler).

Any of the two scripts will start a terminal-based user interface for executing SQL queries.

In order to quit the user interface, use the command '`quit;`' (note that terminating with `Ctrl-C` may lead to information loss, due to abnormal termination of the KVS store) .

The query processor implementation currently supports the following SQL syntax:

```
SELECT
    select_expr [, select_expr ...]
    [FROM table_reference
    [ JOIN table_reference  ON col_name = col_name ]
    [WHERE where_condition]
    [GROUP BY {col_name} ]
    [HAVING where_condition]
    [ORDER BY {col_name}
      [ASC | DESC]]
    [LIMIT { row_count }]
```

The select expression can be a column name, or one of the following functions: `count, avg, sum`.

The following two tables are created and maintained automatically, and can be used for writing SQL queries:
Table **webpages**: This table contains all crawled web pages, and has the following structure:
{
string url: *the url of the webpage as a string*,
string domainName: *the domainName for the webpage as a string,*
double pagerank: *the pagerank computed by WP3 pagerank algorithm as double,*
string body: *the content of the webpage as a string and*
double sentiment: *an overall estimation of the sentiment of webpage's content.*
}

Table **entities**: This table contains information for entities that are extracted from the webpages (e.g., adidas) and has the following structure:
{
string webpageURL: *the url of the webpage that contains the entity,*
string name: *the name of the entity and*
double sentimentScore: *The sentiment for the entity in the webpage*
}

The following sample queries can be used to demonstrate the capabilities of the prototype query processor:

- `SELECT domainName, avg(pagerank) FROM webpages GROUP BY domainName ORDER BY avg(pagerank) DESC LIMIT 10;`
- `SELECT domainName, avg(pagerank), avg(sentimentScore) FROM webpages JOIN entities on url=webpageURL WHERE entities.name like 'adidas' GROUP BY domainName HAVING avg(sentimentScore) > 0.5 ORDER BY  avg(pagerank) DESC;`
- `SELECT count(*) from webpages;`
- `SELECT url,sentiment FROM entities WHERE sentiment>0.4 LIMIT 5;`
- `SELECT domainName, sum(pagerank) FROM webpages group by domainName ORDER BY sum(pagerank) DESC LIMIT 10;`

The prototype implementation has the following limitations:

1. The query processor does not support aliases.

2. In queries with a join, the column of the table found in the `FROM` clause must be the left operand in the equality relation. In the following example, the URL column of the webpages table is the left operand in the relation of the join :
   ```
   SELECT domainName,name FROM webpages JOIN entities on url=webpageURL;
   ```
3. The * wildcard can be used only inside a function. For example, the following query is supported
   ```
   SELECT count(*) from webpages;
   ```
   whereas the following query is not supported
   ```
   SELECT * FROM webpages;
   ```
   As a workaround, user must set the column names explicitly.
4. The free version of the third-party, web-service AlchemyAPI used for extracting the entities and sentiment scores imposes a quota on the number of requests per day and user. Furthermore, the service offers no up-time guarantees. If the quota is surpassed or if the service is down, some sentiments may be set to -2.
5. Robustness: If the query processor terminates unexpectedly (e.g., through a Control-C signal, or due to a network or hardware fault) the hosted segment from the KVS may be lost. Clearly, this can lead to information loss. This limitation will be addressed by replication at the KVS level.

7.2 Configuration

The configuration for the query processor is read by the file processor.properties, which is saved at the root directory. The file includes three parameters:

1. `processorInfinispanConfigFile:` This parameter defines the file used to configure the KVS instance that will be started by the query processor.
   `processorSentimentAnalysisKeyFile:` This parameter is used to specify the file containing the AlchemyAPI key to use.
2. `verbose:` This parameter configures the verbosity of the logging information presented in the querying terminal. `verbose` can be set to true or false.

A sample configuration file follows:

```
processorInfinispanConfigFile=infinispan-clustered-tcp-processor.xml
processorSentimentAnalysisKeyFile=key-processor.txt
verbose=true;
```

8. Conclusions

This deliverable summarizes the progress of WP3 at M12 with respect to the real-time processing platform. A special emphasis is given on the query execution engine, which is the core component of the real-time processing platform, necessary for efficient data retrieval. The deliverable also elaborates on the streaming multiple micro-cloud PageRank algorithm developed in the context of LEADS, since PageRank scores are required by most available ranking functions.

Our effort in the second year of the project will be concentrated on optimizations on the query execution engine specifically focused on the multiple micro-cloud infrastructure of LEADS, as well as on adding fault-tolerance to the engine. The engine will be integrated with the query scheduler for supporting the optimizations, but also with the querying interface of Apatar for enabling a graphical user interface for the inexperienced users. For the experienced users, we will work towards developing or extending a declarative query language that enables a family of more aggressive optimizations. Finally, our work on privacy-preserving query processing for point and range queries will be completed and integrated in the system, enabling LEADS users to efficiently store and retrieve private/sensitive information.

9. References

[PCDG13] S. Papadopoulos, G. Cormode, A. Deligiannakis, M. Garofalakis: Lightweight authentication of linear algebraic queries on data streams. SIGMOD Conference 2013: 881-892.

[ACC+10] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, Boom analytics: exploring data-centric, declarative programming for the cloud, *in Proceedings of the 5th European conference on Computer systems, 2010, pp. 223–236.*

[ACH+11] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, Consistency Analysis in Bloom : a CALM and Collected Approach, *Systems Research, pp. 249-260, 2011.*

[LCM+06] B.T. Loo, T. Condie, M. Garofalakis, D.A. Gay, J.M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe and I. Stoica. Declarative Networking: Language, Execution and Optimization. *ACM-SIGMOD International Conference on Management of Data, Chicago, 2006*

[BCY09] A. Boldyreva, N. Chenette, Y. Lee, and A. O Neill. Order-preserving symmetric encryption. In Advances in Cryptology-EUROCRYPT 2009, pages 224-241. Springer, 2009.

[XYH12] L. Xiao, I.-L. Yen, and D. Huynh. A note for the ideal order-preserving encryption object and generalized order-preserving encryption. 2012.

[YKK12] D. H. Yum, D. S. Kim, J. S. Kim, P. J. Lee, and S. J. Hong. Order-preserving encryption for non-uniformly distributed plaintexts. In Information Security Applications, pages 84-97. Springer, 2012.

[LW12] D. Liu and S. Wang. Programmable order-preserving secure index for encrypted database query. In Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pages 502-509. IEEE, 2012.

[PLZ13] Popa, R.A., Li, F.H., Zeldovich, N.: An ideal-security protocol for order-preserving encoding.

[HILM02] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pages 216-227. ACM, 2002.

[HMT04] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, pages 720-731.VLDB Endowment, 2004.

[BW07] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In Theory of cryptography, pages 535-554. Springer, 2007.

[SBC07] E. Shi, J. Bethencourt, T.-H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In Security and Privacy, 2007. SP'07. IEEE Symposium on, pages 350-364. IEEE, 2007.

[Lu12] Y. Lu. Privacy-preserving logarithmic-time search on encrypted data in cloud. In Proc. of NDSS, 2012.

[Ge09] C. Gentry. A fully homomorphic encryption scheme. PhD thesis, Stanford University, 2009.

[Ge10] C. Gentry. Computing arbitrary functions of encrypted data. Communications of the ACM, 53(3):97-105, 2010.

[CKG98] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. Journal of the ACM (JACM), 45(6):965-981, 1998.

[SiC07] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In Proceedings of the Network and Distributed Systems Security Symposium, pages 2006-06, 2007.

[OG12] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In Financial Cryptography and Data Security, pages 158{172. Springer, 2012.

[Ostr90] R. Ostrovsky. Efficient computation on oblivious rams. In Proceedings of the twenty-second annual ACM symposium on Theory of computing, pages 514-523. ACM, 1990.

[WS08] P. Williams and R. Sion. Usable private information retrieval. In Network and Distributed System Security Symposium, 2008.

[BCG10] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast incremental and personalized PageRank. *Proc. VLDB Endow.* 4, 3 (December 2010), 173-184.

[PLB+99] Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The PageRank citation ranking: bringing order to the web." (1999).

[ALN+07] Konstantin Avrachenkov, Nelly Litvak, Danil Nemirovsky, Natalia Osipova. “Monte Carlo methods in PageRank computation: When one iteration is sufficient”, SIAM Journal on Numerical Analysis, 2007

[LM04] A. N. Langville and C. D. Meyer. Updating PageRank with iterative aggregation. In WWW Alt. ’04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, pages 392–393, New York, NY, USA, 2004. ACM.

[V85] Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37-57. DOI=10.1145/3147.3165 http://doi.acm.org/10.1145/3147.3165

[NRN+10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, S4: Distributed Stream Computing Platform**,** *in 2010 IEEE International Conference on Data Mining Workshops (ICDMW), 2010, pp. 170-177.*

[Sto13] Storm: Available online at https://github.com/nathanmarz/storm/wiki.

[Fet10] Christof Fetzer. StreamMine: a scalable and dependable event processing platform. In Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS '10). ACM, New York, USA, 222-222.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15, 2 (2006), 121-142.

[Apa13] Apatar web site: Available online at http://www.apatar.com/

[BB70] Bloom, Burton H. Space/Time Trade-offs in Hash Coding with Allowable Errors, Communications of the ACM 13 (7): 422–426.

[CM04] Cormode, Graham, S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. J. Algorithms 55: 29–38.

[PGD12] Odysseas Papapetrou, Minos N. Garofalakis, Antonios Deligiannakis: Sketch-based Querying of Distributed Sliding-Window Data Streams. PVLDB 5(10): 992-1003 (2012)

[SSK06] I. Sharfman, A. Schuster, and D. Keren, A geometric approach to monitoring threshold functions over distributed data streams. Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 2006, pp. 301–312.

Appendix A. Publications for LEADS until Month 12

S. Papadopoulos, G. Cormode, A. Deligiannakis, M. Garofalakis: Lightweight authentication of linear algebraic queries on data streams. SIGMOD Conference 2013: 881-892.