



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:	D3.3
Title of Deliverable:	Programming models, querying tools and optimization
Contractual Date of Delivery:	M24 – 2014-09-30
Actual Date of Delivery:	2014-09-29

Abstract

This software deliverable delivers the first version of the programming models, tools, and optimization techniques. With this deliverable, we satisfy the requirements of basic data mining tasks over a mashup-driven querying interface, provide an initial version of the declarative network language, and enable basic query processing over encrypted data. We also include a set of initial optimizations for energy efficiency, scalability, and performance; these include in-situ processing abilities via the plugin architecture, and sketching techniques for compact summarization of big data.

List of Contributors

Name	Organization	E-mail
Ata Turk	BM-Y!	ata@yahoo-inc.com
Diego Marron	BM-Y!	diegom@yahoo-inc.com
Tim Potter	BM-Y!	tep@yahoo-inc.com
Xiao Bai	BM-Y!	xbai@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
Pedro Ruivo	Red Hat	pedro@infinispan.org
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ekaterini Ioannou	TSI	ioannou@softnet.tuc.gr
Eleftherios Chatzilaris	TSI	echatzilaris@softnet.tuc.gr
Evangelos Vazeos	TSI	vagvaz@softnet.tuc.gr
Ioakim Perros	TSI	imperros@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Nikolaos Giatrakos	TSI	ngiatrakos@softnet.tuc.gr
Nikolaos Pavlakis	TSI	npavlakis@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
André Martin	TUD	andre.martin@tu-dresden.de
Christof Fetzer	TUD	christof.fetzer@tu-dresden.de
Do Le Quoc	TUD	do@se.inf.tu-dresden.de
Frank Busse	TUD	frank.busse@tu-dresden.de
Frezewd Lemma Tena	TUD	frezewd_lemma.tena@mailbox.tu-dresden.de
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch
Raluca Halalai	UniNE	Raluca.Halalai@unine.ch
Valerio Schiavoni	UniNE	Valerio.Schiavoni@unine.ch



Document Approval

	Name	Email	Date
Approved by WP Leader	Minos Garofalakis	minos@softnet.tuc.gr	2014-09-30
Approved by GA Member 1	Christof Fetzner	Christof.Fetzner@tu-dresden.de	2014-09-30
Approved by GA Member 2	Etienne Riviere	etienne.riviere@unine.ch	2014-09-30

Contents

LIST OF CONTRIBUTORS	II
DOCUMENT APPROVAL	III
CONTENTS	IV
EXECUTIVE SUMMARY	1
1. INTRODUCTION	2
2. PROGRAMMING MODELS AND TOOLS	4
2.1. DECLARATIVE LANGUAGE.....	4
2.2. PLUGIN ARCHITECTURE.....	7
2.2.1 <i>Motivation and Challenges</i>	7
2.2.2 <i>Overview of the plugin architecture</i>	7
2.2.3 <i>Extending LEADS functionality with plugins</i>	8
2.3 PRIVACY-PRESERVING QUERYING.....	9
2.4 SKETCHES FOR BIG-DATA SUMMARIZATION.....	12
3. QUERYING TOOLS	14
3.1. MASHUP-BASED USER INTERFACE.....	14
3.2. WEB SERVICE AND TERMINAL-BASED USER INTERFACE.....	17
4. OPTIMIZATIONS	17
5. CONCLUSIONS	19
6. REFERENCES	20
APPENDIX A.	22
A.1. PLUGIN ARCHITECTURE.....	22
A.1.1. <i>Plugin Development and deployment</i>	22
A.2. PRIVACY-PRESERVING POINT QUERIES.....	24
A.2.1. <i>Quick Start</i>	24
A.3. MASHUP-BASED USER INTERFACE DESCRIPTION.....	26

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

Executive summary

This deliverable presents our progress on the real-time processing platform since Month 12. This document focuses on three important aspects: a) programming models and tools, b) querying tools, and c) optimizations for performance and energy efficiency.

Programming models and tools. In Section 2 we present our progress with respect to programming models and tools. We start by discussing a simple yet powerful SQL-like declarative language, which is partially implemented in the M24 software deliverable, and will be fully realized until the end of the project. We then describe a new functionality that focus on enabling LEADS expert users to execute arbitrary computations on both streaming and static data. The functionality relies on the plugin architecture that is now part of the real-time platform. Finally, we present LEADS-specific supporting tools that focus on making development for LEADS easier. These tools include support for outsourcing and querying of private/sensitive data in the LEADS platform, as well as a set of sketches for summarizing vast amounts of streaming information.

Querying tools. In Section 3 we discuss our progress towards supporting straightforward and intuitive querying over the LEADS platform. We start by presenting the graphical user interface (GUI) of the processing platform that enables users to retrieve information from the LEADS infrastructure using mashups. The user interface is built over Apatar, a state-of-the-art mashup tool, and therefore has a familiar look-and-feel for existing mashup users. For the expert users, the GUI also enables writing, deploying and executing small snippets of code in the LEADS processing platform. We then briefly describe the enhancements on the web service and the terminal-based querying environment since Month 12.

Optimizations. Section 4 focuses on the optimization-related enhancements of the query processing engine. The main enhancements discussed in this section include a more mature query planner, a better query executor that removes significant memory limitations of the Month 12 version, and integration with the query scheduler for energy efficiency, scalability, and better response times.

Appendix A. This deliverable is a software deliverable. As such, it includes design and usage details for the platform functionality, e.g., algorithms, as well as code samples for efficiently using the offered functionality. Since this is a public deliverable, to keep the main body of the deliverable in a level appropriate for the general audience, we have decided to include details that are not necessary for the majority of readers in an appendix. In the appendix, we also include details for the prototype, such as source code location, access information, and sample usage.

1. Introduction

WP3 is responsible for offering the distributed data processing and querying capabilities of the LEADS platform. This is achieved by interacting with WP2 distributed storage engine, and WP4 scheduler and data placement components. The core functional requirements for the data processing engine are as follows:

1. Querying tools, for users to express their queries using both a declarative language for expert users and a graphical user interface for novice users.
2. Novel programming models and tools to enable the development of the distributed data processing algorithms across multiple micro-clouds.
3. Extending the data processing capabilities of the LEADS platform to facilitate the development of distributed data mining algorithms using domain specific data processing through the support of plugins.
4. Privacy-preserving query capabilities to enable users to store and query sensitive data safely.
5. Optimizations specialized for the unique architecture of the LEADS micro-clouds federation and achieve high performance and energy efficiency.

The purpose of this document is to describe our results (research and implementation) with respect to the aforementioned requirements. The document is structured as follows. In the next section we discuss our progress on programming models, i.e., the support that the LEADS platform provides to the developers for exploiting the platform (hardware resources, functionality, data) to perform arbitrary computation. In this aspect, the M24 deliverable implements a simple, yet powerful, SQL-like declarative language, which can be used to access and manage the data stored in the KVS in a location-transparent manner. Notice that the implementation is not only a proxy to the KVS interface; its expressiveness enables users to execute projections, filters, joins, and other operations on data that may reside across multiple micro-clouds. For performance and scalability, the operators are implemented in a distributed fashion by the LEADS processing platform.

Considering further the programming challenges that the LEADS developers would face, we have included three additional contributions. The first one is the plugin architecture, an architecture enabling any third-party user to plug-in arbitrary code to be executed on the LEADS data - streaming or stored. Many plugins are already integrated in the LEADS platform, e.g., to support distributed sentiment analysis, entity extraction, and keyword-based indexing on the crawled web pages. The second contribution addresses the need to outsource private/sensitive data in the LEADS platform (even in the presence of compromised nodes) and then safely retrieve them. For this, we have implemented a state-of-the-art algorithm for point queries over encrypted data. The algorithm operates across micro-clouds, fully exploiting the capabilities offered by the LEADS storage layer. Finally, the third contribution enables LEADS developers to summarize vast amounts of streaming information through the use of sketches. The contribution is accessible to the LEADS developers through plugins. It can be used to greatly reduce the memory and network resources requirements for processing big data.

In Section 3, we present our efforts to enable straightforward and intuitive querying over the LEADS platform. We start by presenting the LEADS-specific graphical user interface that enables users to retrieve information from the LEADS infrastructure using mashups. The user interface is built over Apatar - a state-of-the-art mashup tool - and therefore has a familiar look-and-feel for existing mashup users. Even though the interface is not fully completed, it already supports a large part of the envisioned operators. Focusing on expert users, the user interface enables writing, deploying and

executing small snippets of code in the LEADS processing platform. This code is typically MapReduce-based, and it is executed next to the data, even across micro-clouds, exploiting the power of the underlying query execution platform. Furthermore, we describe the main enhancements on both the web service and the terminal-based querying environment since Month 12.

In Section 4, we present the key optimizations on the query processing engine that were implemented for increasing the energy efficiency of the system, as well as reducing the latency. These optimizations include a more mature query planner, a better query executor that removes significant memory limitations of the version presented in D3.2 at Month 12, and integration with the query scheduler.

This deliverable is a software deliverable. As such, it includes design and usage details for the platform functionality, e.g., algorithms, as well as code samples for efficiently using the offered functionality. Since this is a public deliverable, to keep the main body of the deliverable in a level appropriate for the general audience, we have decided to include such details that are not necessary for the majority of readers in Appendix A. Access details for the M24 prototype, sample functionality, as well as source code location are also provided in the appendix.

2. Programming models and tools

In this section we describe the current development and planned future work with respect to the LEADS programming models.

2.1. Declarative Language

One of the primary requirements of LEADS is the support of a declarative language for inserting, organizing, and retrieving both static and dynamic (streaming) information. To address this requirement, we offer the *abstraction of a standard relational database* over the LEADS platform, which can be manipulated with an SQL-like declarative language. SQL was selected as the basis due to its wide acceptance and maturity, and extended with a simple syntax in order to handle data streams. Unlike previous cloud-based relational database implementations, e.g., Impala¹ and Hive², LEADS tables can span multiple micro-clouds, utilizing the scalability properties of Ensemble caches (see F11 in D2.4). Furthermore, we extended SQL to include version-based window functionality over the data stored in Ensemble caches, utilizing the versioning functionality implemented in WP2 (see C61, F61 in D2.4). The version-based window takes as parameters a starting boundary T1 and an ending boundary T2, and outputs the portion of data inside the interval of T1-T2.

In addition to SQL functionality on stored data, we also need to support operations on data streams. Data streams are inherent to the LEADS environment and service model. For example, an event stream can be created by monitoring the KVS for insert/update/delete operations (see section 2.2 for a discussion on how such streams could be created), or even by directly monitoring external data feeds (e.g., financial data streams), with arbitrary code supplied by the user. Since original SQL is focused on static data, we extended the language to enable querying over a combination of streaming and static data. Precisely, we included time-based sliding window functionality over data streams, as supported by the Continuous Query Language [ASJ03]. A time-based sliding window on a stream S takes a time-interval T as a parameter and is specified by following the reference to S with [Range T]. Intuitively, a time-based window defines its output relation over time by sliding an interval of size T seconds capturing the latest portion of an ordered stream.

¹ <http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html>

² <https://hive.apache.org/>

The LEADS SQL syntax is as follows:

Catalogue management:

Users can create a new table as follows:

```
CREATE TABLE table_name (column_name data_type, ... )
```

Supported data types are:

bool, int, float, double, text, timestamp

Type	Description	Example
bool	Boolean value	true/false
int	integer value (32bit)	-2^{31} (-2,147,483,648) to $2^{31} - 1$ (2,147,483,647)
float	single-precision real number (32bit)	
double	double-precision real number (64bit)	
text	Unicode text	
timestamp	YYYY-MM-DD hh:mm:ss	timestamp '2013-12-17 12:10:20'

Table names and column names must have the following form:

```
table_name: identifier  
column_name: identifier  
column_names: column_name [, column_name ]*  
identifier: ['a'..'z'|'A'..'Z'|'_' ] ['a'..'z'|'A'..'Z'|Digit|'_' ]*
```

Tables can be deleted as follows:

```
DROP TABLE table_name
```

Tables can be altered as follows:

```
ALTER TABLE table_name [ RENAME TO table_name | RENAME COLUMN column_name TO  
column_name | ADD COLUMN field_element ]
```

Indexes are structures that speed-up the access time on stored data and they can be created as follows:

```
CREATE INDEX identifier ON table_name (column_names)
```

Indexes can be deleted as follows:

```
DROP INDEX identifier
```

Data management and retrieval:

A record can be inserted into an existing table as follows:

```
INSERT INTO table_name (column_names) VALUES(value1,value2,value3,...);
```

For updating existing records:

```
UPDATE table_name SET column_name1 = value1, column_name2 = value2,... WHERE  
condition;
```

Condition is a combination of one or more predicates that use the logical operators **AND**, **OR**, and **NOT**:

```
Condition: [ NOT predicate | (condition) [AND | OR] [ NOT ] [ predicate |  
(condition) ] ]
```

```
predicate: expression [ = | <> | != | > | >= | < | <= ] expression  
| column_name [ NOT ] LIKE pattern
```

```
expression: column_name | constant | function(column_names)
```

```
function: COUNT, MIN, MAX, AVG, SUM
```

```
pattern: Is a string inside single quotes of characters and the wildcard character  
'%'. The percent sign represents Any string of zero or more characters.
```

Records can be deleted as follows:

```
DELETE FROM table_name WHERE condition;
```

Records can be retrieved as follows:

```
SELECT * | select_list  
[ FROM reference1 JOIN reference2 ON column_name1 = column_name2 ]  
[ WHERE condition ]  
[ GROUP BY column_name ]  
[ HAVING condition ]  
[ ORDER BY column_name [ ASC | DESC ] ]  
[ LIMIT row_count ]
```

```
select_list: [[column_name | function(column_names)] [AS alias]? ] [, select_list ]*
```

```
reference: [table_reference | stream_reference] [AS alias]
```

```
table_reference: table_name | VERSION [from_timestamp:to_timestamp] table_name
```

```
stream_reference: stream_name | RANGE T stream_reference
```

With table_reference we define a table that exist in our database (including version constraints) and with stream_reference we define an incoming stream for time-stamped tuples (including sliding window constraints).

The current implementation of the query execution engine supports the commands of create table, drop table, select (with all described operators) and insert. For scalability purposes, implementation of most operators is based on the mutli-cloud MapReduce model described in Section **Error! Reference source not found.** All the operators are built on top of Ensemble caches (D2.4). Besides enabling exploitation of a much larger resource pool, this also enables moving the execution of

operators next to the data, which substantially reduces the network requirements. This is particularly the case for joins, functions, and conditions.

Versioning and time-range constraints on select statements, as well as deletes and updates, alter table, drop table, create index and drop index are still not fully supported. These commands will be implemented in the third year of the project, possibly utilizing the functionality offered by the storage layer, e.g., for indexing, we may extend the single-cloud indexes offered by WP2 (C61, D2.4).

2.2. Plugin architecture

A crucial property of LEADS is its extensibility in order to cover user requirements and to address arbitrary processing needs. Users can extend LEADS functionality through the use of plugins. In this section, we first discuss the motivation and challenges behind developing a multi-cloud plugin-based execution engine. Then, we present a high-level overview of the LEADS plugin system, and outline the process of developing and incorporating new plugins. Finally, we briefly present the plugins that have been already implemented and integrated in the platform.

2.2.1 Motivation and Challenges

Declarative languages such as SQL/CQL are powerful tools for expressing complex queries over stored/streaming data; however, they provide limited capabilities for extending the processing and mining functionality of the system. Thus, over the years, the major Relational Database Management Systems (RDBMS) enabled their users to extend their functionality. Towards this direction, LEADS enables users to plug-in their own code that can be invoked at will, i.e., after each insert or update (similar to the triggers functionality at standard relational databases).

Developing a plugin architecture in the LEADS settings entails several non-trivial technical challenges. In the vast majority of cases, a table in LEADS is partitioned and distributed across multiple micro-clouds. Developing and deploying a plugin that monitors all partitions of such a table should be transparent to the user, i.e., as if the table was centralized. Thus, the LEADS platform should be able to handle changes in the partitions, e.g., if a new micro-cloud is assigned part of the table for load-balancing purposes. Our plugin architecture addresses these issues by handling everything related to the deployment and management of plugins in LEADS automatically, i.e., installation of plugins source code to new micro-clouds, so that the plugins are executed locally, next to the data, and subsequently tracking of the events that occur in these micro-clouds by utilizing the listeners functionality offered by the storage layer (C12, F12 in D2.4).

Before going into further details, it is important to note that plugins are reusable components that can be combined to create sophisticated processing workflows to enable users implement more complex processing tasks. For example, a user may want to extract from e-commerce websites information for various products, such as name, price, and reviews, and then maintain the top-k products in terms of price. Such a workflow can be implemented by developing two plugins, one for the extraction of required attributes (products, price, reviews) from e-commerce sites, and another for maintaining the top-k products in terms of price. Additionally, a user can further extend his/her workflow with additional stages at a later stage.

2.2.2 Overview of the plugin architecture

Plugins in LEADS are self-contained data processing units that are triggered when there is a table change, i.e., a record creation, modification or removal. As such, each plugin needs to be attached to

a target relation table. Figure 1 depicts the LEADS plugin architecture. The architecture is composed of three main components and two KVS caches for storing all the installed plugins and system-related meta-data. The System Plugin Repository maintains all the installed plugins of the LEADS platform, which have been uploaded by the users. The Active Plugin Repository maintains deployment meta-data for the deployed plugins.

The Plugin Manager component is responsible for the deployment and undeployment of each plugin. The component reads the plugin (code and configuration) from the System Plugin Repository cache, stores the plugin-related data (source code and configuration) to the Active Plugin Repository, and creates a Plugin Handler instance to handle the actual deployment to all nodes that contain partitions of the target table. The Plugin Handler first installs a specialized listener (C12, F12 in D2.4) that enables us to seamlessly track changes of the micro-clouds topology, and initialize the Plugin Runner component at the dynamic set of nodes of the Ensemble cache. In turn, Plugin Runner at each node loads the plugin code, and starts listening for the selected events (e.g., deletions) that occur on the partition of the target table stored at the node. In the following paragraphs, we provide more details on the process of developing and using the plugins.

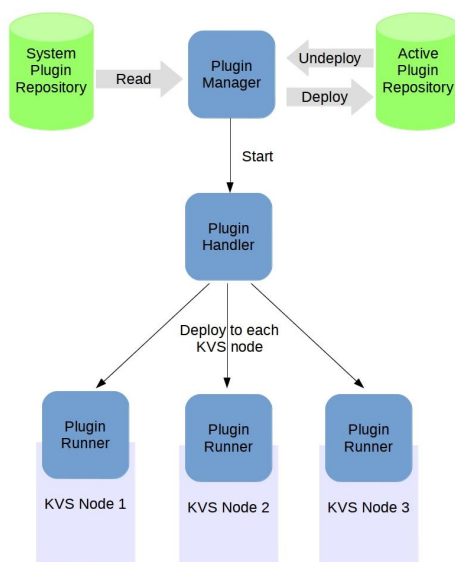


Figure 1. High level overview of the LEADS plugin architecture

2.2.3 Extending LEADS functionality with plugins

Developing a new plugin is a straightforward procedure. Briefly, the developer only needs to implement the plugin interface (see Appendix A.1), and export the bytecode to a jar, together with all dependencies (e.g., third-party libraries invoked from the plugin). Then, the user uploads the jar to the platform using the uploadPlugin method from the provided client library (for technical details read Appendix A.1), together with any configuration that might be needed by the plugin, saved as an XML file. The plugin can be deployed using another library call (deployPlugin), by defining the id of the plugin, the target table, and the type of events for which the plugin should be invoked (insert,

remove, or update). Further technical details together with code examples of the aforementioned procedure are included in the Appendix A.1.

Our previous discussion focused on plugins that are triggered by updates on selected tables. It may frequently be required, however, to implement complex workflows on feeds of external streams. To enable such functionality, LEADS users can develop plugins that monitor external sources of events and write these events to an auxiliary table. Then, arbitrarily complex workflows on the stream data can be constructed by simply attaching plugins to this auxiliary table.

Implemented Plugins

The plugin architecture is also used internally in the project in order to implement functionality requested by the end-user (adidas). In particular, the key plugins currently installed in LEADS are:

Entity Recognition Plugin. The entity recognition plugin receives as input a web-page and extracts the entities that are present in the body of the web-page. All the extracted entities are written to a table.

Sentiment Analysis Plugin. The Sentiment analysis plugin receives as input a web-page and performs sentiment analysis to the overall body of the page. Combined with the previous plugin, this plugin can also compute the sentiment for all entities identified in each web-page. Then, the plugin updates the table created by the Entity Recognition plugin with the computed sentiments. For performance reasons (to require a single parsing of the web-page), the Entity Recognition and Sentiment Analysis plugins are in practice combined to a single implementation.

PageRank Plugin. The plugin implements a novel fully distributed PageRank maintenance algorithm, developed particularly for the LEADS infrastructure. A discussion of this algorithm can be found in the past deliverable D3.2.

Furthermore, a plugin for top-k maintenance is currently under development. The plugin implements the algorithm of Babcock and Olston [BO03], and could be used, for example, for maintaining a list of the k most influential web-pages (i.e., the pages with the highest PageRank score).

2.3 Privacy-preserving querying

Over the years the problem of privacy-preserving querying has emerged in the Crypto and DB communities. The importance of the problem has been further elevated with the recent popularity of data and computation outsourcing, e.g., under the Data-as-a-Service and Infrastructure-as-a-Service models. In such cases, privacy is essential due to the presence of adversaries. Potential adversaries may be the cloud service provider itself, its users and insiders, and any third party attackers who are capable of viewing the target data, monitoring query processing on the data, obtaining or inferring data and queries, as well as gaining unrestricted access to the cloud servers. For our discussion, we assume that the adversary is the service provider itself (untrusted server), which is the worst case and the most demanding scenario - any solution proposed for this case fully addresses the other threat models as well. This problem is highly related to LEADS, since many third-party micro-clouds (possibly untrusted or competitive) comprise a LEADS working installation. In this section, we explain how we enable privacy-preserving point queries over the LEADS platform. Point queries were selected for this first contribution, since they are the core query type for data retrieval at all database management systems (both centralized and distributed).

We start with a motivating example. Let us assume a database that contains the relational table of Table 1, which includes personal details of users that need to be protected. Our goal is to outsource

this database over LEADS without revealing/leaking any of the private data contained in the table to unauthorized parties. At the same time, we want to enable point queries on some preselected columns. For example, we might want to be able to retrieve all persons with a given salary. Privacy-preserving query execution is required for addressing such requirements.

Table 1

Id	Name	Job	Salary
1	John	Artist	1k
2	Bob	Singer	2k
...
N	Alice	Actress	1k

A naive solution to achieve privacy preserving point querying is to have the client (the owner of the data) encrypt the whole database using a powerful encryption scheme, prior uploading it to the cloud. Each time a point query arrives, the client downloads the entire encrypted database from the cloud, decrypts the database locally, and executes the query. To encrypt the database, we can use randomization schemes which are CPA-secure schemes (Chosen Plaintext Attack), also known as RND schemes. Intuitively an encryption scheme is CPA-secure if it outputs encrypted messages (ciphertexts) that do not reveal **any** partial information about the original messages (plaintexts) (see [KL08] for further details). Although this simple solution achieves the desirable strong security level, it also requires extremely high communication cost for downloading the whole database at query time. Therefore, this solution is completely impractical.

A more efficient solution is to use Property Preserving Encryption (PPE) schemes, which encrypt data in a way that only a certain part/property of the underlying information is leaked. There are different types of PPE schemes, with the most common being the Deterministic Encryption scheme (DET). An encryption scheme is DET if it always encrypts the same message (plaintext) to the same ciphertext. DET encryption can be used to enable point queries in a privacy-preserving manner, by checking if two encrypted messages are equal, i.e., it works directly on the ciphertexts, without needing to decrypt them.

Clearly, DET schemes leak the equality property, i.e., a malicious or curious participant/cloud node will know that two encrypted messages are the same if they have the same ciphertext. When this leakage is acceptable, DET schemes can be combined with randomized encryption schemes (RND) to support privacy-preserving point queries as follows. The DET scheme is used to encrypt the tuple attributes on which we want to be able to execute privacy preserving point queries (the salaries, in the example of Table 1), whereas with the RND scheme we encrypt the remaining attributes (id, name, and job). Both these encryptions take place on trusted resources, e.g., on the machine of the data owner. Once the encryption is over, the encrypted database (EDB) is uploaded to the cloud. In order to query the EDB, we encrypt the queried value (i.e., the salary for which we want to retrieve all persons) with the DET scheme, using the same encryption key as the one used for the respective attributes in the encryption phase. Then, we send the encrypted value to the cloud, so that each cloud server can examine each tuple in the EDB and find those that have the same encrypted value as the queried ciphertext. The tuples that satisfy the above condition are returned to the client and decrypted. Therefore, the desirable efficiency is achieved by pushing to the cloud machines the responsibility of checking if an encrypted value in the EDB is part of the query answer. Furthermore, this approach drastically reduces network requirements compared to the first CPA-secure method,

since now only the encrypted values that satisfy the query need to be sent over the network. This also translates to huge energy savings, since network usage is typically responsible for a large part of data centers energy consumption. Further optimizations, e.g., exploitation of indexes can easily be integrated to speed-up DET schemes and reduce computational and network complexity.

From the perspective of security, as discussed earlier, the solution leaks the equality property, leading to a severe consequence in which the server can learn the frequency with which a tuple is assigned a certain value (for a specific attribute). A direct consequence of the equality attack is that the EDB is vulnerable to frequency analysis attacks. In other words, for the example of Table 1, if an attacker knows that salaries follow a particular distribution, she will be able to map each encrypted value to a salary. This problem is further aggravated when the data encrypted with DET (the salaries in the running example) follows a distribution with high skew, e.g., the power law distribution.

Despite these vulnerabilities, the above solution is widely used by the DB community in order to support privacy preserving point queries in an efficient manner. Two such state-of-the-art approaches, are CryptDB [TKM+13] and Monomi [PRZ+11]. Both are vulnerable to statistical attacks, since they leak the equality property, and thus achieve weaker security guarantees.

Clearly, there is a trade-off between efficiency and security, and DET and CPA-secure lie at the two extremes (Figure 2). DET offers high efficiency by revealing the equality property, and thereby making the data susceptible to statistical attacks, whereas CPA-secure is fully secure but has huge network and computational overhead for executing each query.

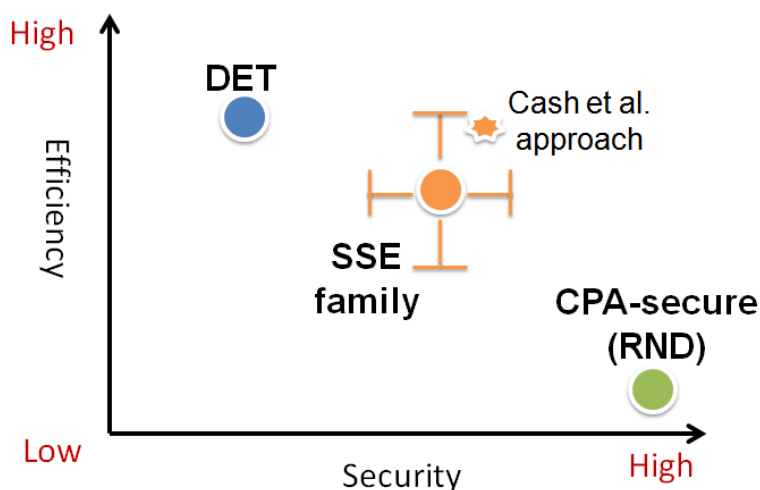


Figure 2. Security Vs Efficiency trade-off

To cover the area between DET and CPA-secure, the Crypto community has proposed a family of approaches called Searchable Symmetric Encryption (SSE) schemes [CM+05, CGK+06-SSE1, CGK+06-SSE2, CK10, KPR12, CJJ+13]. SSE schemes provide solutions to the problem of privacy-preserving keyword search, which is very similar to our problem of privacy preserving point querying. More specifically, these schemes allow the data owner to securely encrypt and outsource the data, and permit query processing directly on the ciphertexts. Most of these works focus on keyword queries, which are very similar to the privacy-preserving point queries: “keywords” corresponds to the column of the values on which we will execute point queries (searchable values, the salaries in the

previous example) whereas “documents” corresponds to the tuples associated with the searchable value (the remaining attributes in our example). This mapping allows us to exploit the SSE schemes in order to leverage the gap between efficiency and security that we encountered in previous solutions. The high-level idea is as follows. Offline, a client produces ciphertexts by encrypting her data with a secret key sk and uploads the ciphertexts c to an untrusted server. She also creates and sends a secure index I on the data for efficient keyword search, e.g., an index on the salary column. For executing a query, given a keyword w , the client generates a query token t_w using the same key sk that was used to encrypt I . The server uses t_w on I to retrieve the IDs of the ciphertexts associated with w . The results c_1, \dots, c_m are retrieved and transmitted back to the client, who eventually decrypts them with her key.

From the perspective of security, state-of-the-art approaches offer either the **non-adaptive** or the **adaptive** security guarantees [CGK+06]. An intuitive explanation of the non-adaptive definition is that it guarantees the security of a scheme assuming that the client generates all queries at once, i.e., in a single batch. Since, in practice, clients frequently want to perform exploratory/subsequent queries, a non-adaptive guarantee has a rather limited practical applicability. Adaptive guarantees overcome the above limitation by enabling the execution of subsequent queries without leakage. With respect to efficiency, some approaches achieve optimal search time, equal to simple (non-privacy preserving) keyword search problems [CM05, CGK+06-SSE1, CGK+06-SSE2, LSD+10, CK10, KPR12, CJJ+13]. Among these, [CGK+06-SSE2, LSD+10, CK10, KPR12, CJJ+13] achieve the adaptive security definition. The ones of [KPR12, CJJ+13] are of particular interest, since these have storage requirements linear to the size of the document collection.

Having in mind both the security and efficiency characteristics of these approaches, we have selected to integrate in LEADS the approach by Cash et al. [CJJ+13]. The approach guarantees the adaptive security definition, and at the same time achieves the best storage and search performance of all existing adaptive SSE protocols. In Appendix A.2, we provide more implementation details about the implemented algorithm and we give instructions and a detailed example of how the scheme can be used within LEADS.

2.4 Sketches for big-data summarization

In the context of modern cloud-based applications, the ability to summarize vast amounts of data (either dynamic or static), and answering queries by using only the summary, is frequently required. This requirement becomes even more important in LEADS, since LEADS developers might frequently need to perform processing of data across micro-clouds, without shipping the full data over the network. Sketches are a family of data structures that enable such summarization. To offer this ability to LEADS developers, we have implemented the most popular sketches in LEADS as plugins. The implemented sketches can be used to answer the following queries:

Membership queries: e.g., does a data set/stream contain item X . To handle this type of query, we offer sketches from the Bloom filter family, i.e., original Bloom filters, Counting filters, and Bloom filter extensions {B70, FCA+00, PGD12}.

Set cardinality estimation: e.g., how many distinct items are contained in a data set/stream. We support this functionality using Bloom filter extensions [PSN10].

Count of arrivals over sliding windows: e.g., how many items arrived in the last 10 seconds. The state-of-the-art data structures for supporting these queries are exponential histograms [DGI+02] and randomized waves [GBT02]. Both are integrated in the project.

Frequency counts: e.g., how many times is item X contained in a data set/stream. These are by far the most frequently requested queries. We support them with Count-min sketches [CM05] and ECM-sketches [PGD12] (ECM-sketches provide the same functionality over a sliding window constraint).

Self-join and inner-product size estimation: e.g., what is the self-join size of the item frequencies in a data set/stream. This query type is typically used for constructing efficient query plans in relational databases, and will also be required for the full implementation of our query planner. The supporting sketches integrated in LEADS are Count-min sketches, AMS sketches [AMS96], and ECM-sketches (ECM-sketches provide the same functionality over a sliding window constraint).

The detailed API of the described sketches and usage examples are available in the following URL: <http://www.leads-project.eu/knowledge/Sketches>.

3. Querying tools

3.1. Mashup-based user interface

The M12 prototype supported only SQL queries through a command-line terminal. Now, novice users can also retrieve information from the LEADS platform without writing code, with the use of a simple and intuitive graphical user interface based on mashups.

Many mashup tools have been developed to support creation and execution of consumer-focused and enterprise mashup applications. We have chosen Apatar for several reasons, with the most important being the source code availability. Apatar is open source software, distributed under the GNU General Public Licence (GPL). Note that Apatar is one of the very few options that offer source code availability, thereby enabling extensions. For example, the also mature tool Yahoo! Pipes is closed-source, and not extensible. Another candidate open-source tool was Taverna³, but is currently not as mature as Apatar.

The graphical user interface relies on an Apatar extension. The core functionalities of the tool are represented as connectors. There exist connectors to perform data loading (e.g., to retrieve data from a relational database), integrations (e.g., to perform joins between two tables), and transformations (e.g., to perform basic text processing). Constructing an ETL process is as simple as selecting the appropriate connectors from an existing library, parameterizing them, and linking them together to form a mashup.

In the context of LEADS, we have only used the core Apatar GUI (without the connectors) since the default Apatar connectors (as well as their centralized implementations) were irrelevant to the LEADS platform. To enable novice users to fully exploit the LEADS functionality, we needed to implement the following LEADS-specific connectors:

- **Read:** The Read connector specifies the input table. All workflows must have at least one Read Connector, for feeding the data.
- **Output:** The Output connector points to a table for writing the results of the workflow. If there is already a table with the same name, the user can override the data stored within the table with the new data. Any workflow must have one or more configured Output Connectors.
- **Filter:** The Filter connector filters the input data. We have implemented numerous graphical sub-connectors that allow the user to filter the input data, in a similar manner to the "where" clause in an SQL like language.
- **Sort:** The Sort connector is used to sort the input data according to one or more attributes. The connector receives as input the results of another connector, e.g., a read connector, and outputs the same data sorted on the selected attribute(s).
- **Grouping:** The Grouping connector implements the Group By functionality of standard SQL. The connector allows the user to define a set of group attributes, the aggregation functions and the output attributes.
- **Limit:** The Limit connector implements the limit functionality of standard SQL. The connector receives the results of another connector as input, and outputs only the records that satisfy the limit constraint to the output.

³ <http://www.taverna.org.uk/>

- **Project:** The Project connector receives the results of another connector as input, and outputs only a subset of the attributes.
- **Distinct:** The Distinct connector receives the results of another connector as input, and outputs only the distinct values of a preconfigured attribute.
- **Equi-Join:** The Equi-Join Connector receives the results of two other connectors as input, and performs the join on a selected attribute.
- **MapReduce:** The MapReduce Connector enables integrating arbitrary MapReduce functionality in a workflow. It allows the user to choose the path location of the jar file, the Mapper's and the Reducer's class names, and the path of the corresponding configuration file. In order to be utilized in complex mashups, the connector also allows the user to specify the output attributes.

So far, all but the Distinct and MapReduce connectors have been fully implemented and integrated in the platform. The implementation of each connector is based on the multi-cloud MapReduce model to achieve scalability and efficiency (section **Error! Reference source not found.**). Further details about the aforementioned connectors, as well as usage examples, are presented in Appendix A.3.

After designing a workflow as an Apatar mashup, the user has the option to save it locally for future use, and/or deploy it in the LEADS platform. Deployment sends the workflow to the query execution engine. The workflow then goes through the standard query planning and scheduling phases, and gets executed at the micro-clouds suggested by the query scheduler. When the workflow is completed, the results are returned to the user.

To illustrate the process, consider the following SQL query:

```
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;
```

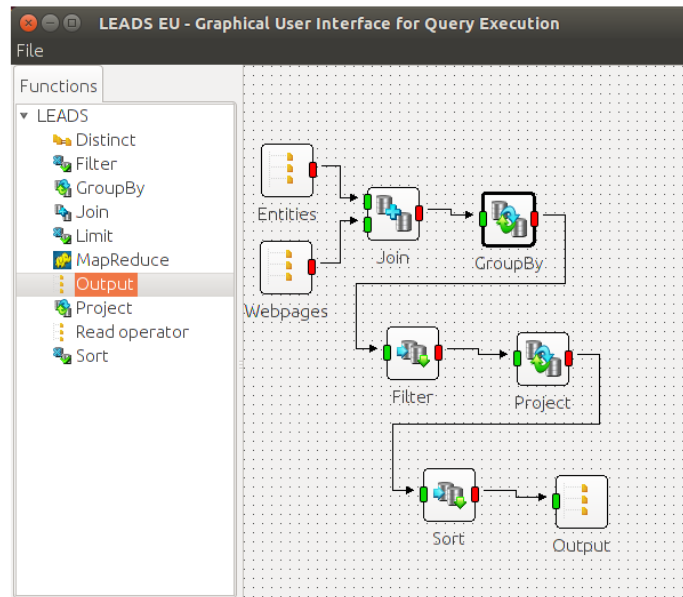


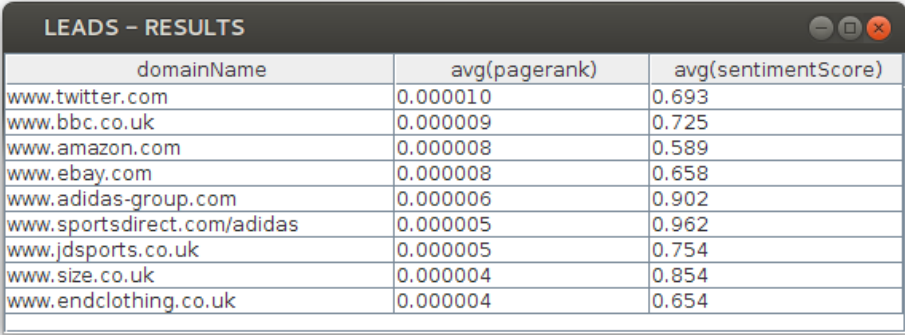
Figure 3. A query workflow

The process of expressing a query such as the above in Apatar is as in Figure 3. First, we drag-and-drop and configure the appropriate connectors from the function area into the empty canvas. We can configure a connector by right clicking on it and by choosing the configure option. Then we wire the connectors, as shown in Figure 3.

In more details, for this example we need to add and configure the following connectors:

- We add and configure the read connectors, by selecting tables Entities and Webpages. Notice that we do not need to provide further configuration details on the actual location (the micro-clouds) of the table; this is handled automatically during query execution.
- We connect the two read connectors to a join connector. We configure the join connector by selecting the join attribute.
- We add a group by operation. For this, we need to configure the grouping attribute (i.e., the domain name), and the desired aggregators - in this case, avg(sentimentScore) and avg(pagerank).
- We filter the data by adding the two conditions of the SQL query to the configuration of the filter connector.
- We configure the project connector in order to keep only columns domainName, avg(pagerank) and avg(sentimentScore).
- We add a sort connector, and configure it by choosing the sorting attribute and the sorting order (descending or ascending).
- We configure the output connector by selecting a cache name, which defines the location where the output data will be stored.

Finally, the results produced by the above query are returned to the user and displayed as shown in Figure 4.



domainName	avg(pagerank)	avg(sentimentScore)
www.twitter.com	0.000010	0.693
www.bbc.co.uk	0.000009	0.725
www.amazon.com	0.000008	0.589
www.ebay.com	0.000008	0.658
www.adidas-group.com	0.000006	0.902
www.sportsdirect.com/adidas	0.000005	0.962
www.jdsports.co.uk	0.000005	0.754
www.size.co.uk	0.000004	0.854
www.endclothing.co.uk	0.000004	0.654

Figure 4 Sample results of Apatar workflow

A detailed description of each connector, as well as more query examples, can be found in Appendix A.3.

3.2. Web service and terminal-based user interface

The LEADS Query Engine enables users to take full advantage of its capabilities in external applications by exposing its functionalities through a RESTful API. The API is already extensively used by other LEADS components, e.g., the Web-Graph Service, the Apatar GUI, and users' plugins.

Our current API supports the following functionalities:

1. Submitting SQL and CQL queries.
2. Inserting and retrieving objects from a table.
3. Submitting a workflow query (Apatar format)
4. Support to specific Web Graph Service queries
5. Getting the status of a submitted query
6. Getting the query results
7. Deploying a plugin

LEADS developers exploit the API to implement sophisticated data processing applications that accommodate their business requirements. To facilitate the development process of those applications, we also provide a Java client library that acts as a proxy to the API, handling the web-service invocation. Furthermore, this library enables some additional functionality, such as uploading a new plugin to the system, encrypting and uploading a dataset to the LEADS platform, and submitting a privacy-preserving point query.

Finally, we have also made minor enhancements to the terminal-based interface by improving the robustness and performance of our implementation.

4. Optimizations

A significant part of the innovation behind the LEADS query engine involves optimizations for performance, scalability, and energy efficiency. The optimizations added since M12 prototype can be broadly classified in the following categories: a) a query planner, to generate efficient distributed query execution plans, b) interaction with the query scheduler, for scheduling and load management across micro-clouds, and finally, c) optimizations on the implementation of the distributed operators. All optimizations target at reducing the network and computational requirements, thereby increasing the energy efficiency and reducing the response time of the platform. We now discuss these optimizations in more details.

Query planning. The LEADS query planner is based on the Apache Tajo query planning engine.⁴ Tajo is an open-source distributed warehouse system, i.e., it implements a full relational database functionality on top of the Hadoop ecosystem. For the purpose of LEADS, we used only the query planner from Tajo, which enables rule-based query optimization (clearly, the operators' implementations of Tajo are irrelevant to LEADS, since LEADS is not built on top of Hadoop). The planner was fully integrated in LEADS and adapted to handle the non-SQL LEADS operators (e.g., for handling complete Apatar workflows, MapReduce tasks, and the relevant queries for the web graph service and for the privacy-preserving point queries). An important property of the planner is that it is extensible, enabling users to modify the planning behaviour. Since the original Tajo planner (and our current implementation) is not oriented towards multi-cloud architectures, we expect that we will be able to achieve substantial performance gains by implementing such extensions. We are currently in the process of implementing two such extensions: a) a multi-phase optimization, where the planner will closely interact with the scheduler to progressively improve the chosen query plan, and, b) a cost-based optimization to reduce the data volume that needs to be transferred across micro-clouds.

Interaction with the query scheduler. The role of the query scheduler is to assign operators to micro-clouds, in order to efficiently utilize the available resources and increase performance and energy efficiency of the platform. Towards this end, the query execution engine annotates the plans generated from the query planner with information that enables the query scheduler to estimate the cost of each possible assignment of operators. In particular, for each operator we provide the following data (for details on how these parameters are used see D4.2):

- **k:** the expected number of VM hours required to process one GB of input data;
- **v:** the expected volume of data transmitted per VM hour, in GB;
- **input cache name:** the name of the input cache for each operator;
- **output cache name:** the name of the output cache for each operator.

For computing k and v , we currently use historical data on the running time of the operators, as well as the input/output ratio. (We are also exploring the use of more advanced properties of the data and the operators, such as cache statistics maintained by the storage layer, complexity analysis results for the operators, and selectivity estimates for the join operator.) The annotated plans are then shipped to the scheduling component, where an assignment of operators to micro-clouds is proposed. The proposed assignment is passed to the query execution engine and executed.

Improvements on the implementation of operators. We also improved the performance of the operators' implementations. In particular, join and sort operators have been re-implemented to minimize the communication cost between the nodes inside a micro-cloud, by enabling in-situ processing, i.e., pushing the computation to the nodes inside the micro-cloud that actually hold the data (recall that the assignment of resources proposed by the scheduler is at the micro-cloud level, and not at the level of individual nodes). As a result, the efficiency of these operators was substantially increased. Additionally, in the M24 prototype, most of our operators have been improved in terms of memory requirements by moving Infinispan caches from main memory to the filesystem. We have also implemented some complex MapReduce operators that group more than one operators of a plan into a single operator. There is a direct correlation between the number of the operators that are executed and the data transferred inside a micro-cloud, since each MapReduce operator reads/writes data from/to KVS. Therefore, the query execution engine groups

⁴ <http://tajo.apache.org/>

multiple MapReduce operators into a single MapReduce operator whenever possible, thereby reducing both execution time and network requirements of the operators.

5. Conclusions

This deliverable summarized our progress in WP3 until M24. As D3.3 is software deliverable, this document mostly serves as a documentation of the functionalities supported by the current query execution engine implementation, and our immediate plans. Our discussion focused on the advancements in programming models and tools, querying tools, and finally, on the developed optimizations of the query execution engine for performance and energy efficiency. We also highlighted the challenges (both technical and research) behind each functionality, and presented the solutions that we have selected for integration in the project. To keep the main discussion as concise as possible, implementation details and usage examples were included in Appendix A.

Entering the third and final year of the project, our work will be focused more on completing the integration with the other work-packages, and on performing further performance and energy optimizations. Furthermore, we will be working more on the multi-cloud related optimizations, in close collaboration with the other work-packages, and particularly with WP2 and WP4.

6. References

- [FCA+00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* 8, 3 (June 2000), 281-293. DOI=10.1109/90.851975 <http://dx.doi.org/10.1109/90.851975>
- [B70] Burton Bloom. "Space/time trade-offs in hash coding with allowable errors". *Commun. ACM*, 13 7_:422-426,1970.
- [KL08] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, Boca Raton, FL, 2008
- [PRZ+11] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty- Third ACM Symposium on Operating Systems Principles*, pages 85-100. ACM, 2011.
- [TKM+13] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases*, pages 289-300. VLDB Endowment, 2013.
- [CM05] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, pages 442-455. Springer, 2005.
- [CGK+06] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 79-88 ACM, 2006
- [LSD+10] P. Van Liesdonk, S. Sedghi, J. Doumen, P. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Secure Data Management*, pages 87-100. Springer, 2010.
- [CK10] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology-ASIACRYPT 2010*, pages 577-594. Springer, 2010.
- [KPR12] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of 2012 ACM conference on Computer and communications security*, pages 965-976. ACM, 2012
- [CJJ+13] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly- scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology-CRYPTO 2013*, pages 353-373. Springer, 2013.
- [ASJ03] Arasu, Arvind; Babu, Shivnath; Widom, Jennifer. *The CQL Continuous Query Language: Semantic Foundations and Query Execution*.
- [BO03] Babcock, Brian, and Chris Olston. "Distributed top-k monitoring." *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003.
- [PSN10] Papapetrou, Odysseas, Wolf Siberski, and Wolfgang Nejdl. "Cardinality estimation and dynamic length adaptation for bloom filters." *Distributed and Parallel Databases* 28.2-3 (2010): 119-156.
- [DGI+02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '02)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 635-644.

- [GBT02] Gibbons, Phillip B., and Srikanta Tirthapura. "Distributed streams algorithms for sliding windows." Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. ACM, 2002.
- [PGD12] Papapetrou, Odysseas, Minos Garofalakis, and Antonios Deligiannakis. "Sketch-based querying of distributed sliding-window data streams." Proceedings of the VLDB Endowment 5.10 (2012): 992-1003.
- [CM05] Cormode, Graham, and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications." Journal of Algorithms 55.1 (2005): 58-75.
- [AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The space complexity of approximating the frequency moments. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC '96). ACM, New York, NY, USA, 20-29. DOI=10.1145/237814.237823 <http://doi.acm.org/10.1145/237814.237823>
- [WTR+12] Lizhe Wang, Jie Tao, Rajiv Ranjan, Holger Marten, Achim Streit, Jingying Chen, Dan Chen, G-Hadoop: MapReduce across distributed data centers for data-intensive computing, Future Generation Computer Systems, Volume 29, Issue 3, March 2013, Pages 739-750

Appendix A.

We will now present technical details and step-by-step instructions on the implementation of new LEADS Query Engine and their integration in the platform. The code of M24 prototype can be found in <https://github.com/vagvaz/Leads-QueryProcessor>, whereas access details for the demo can be found in <http://www.leads-project.eu/knowledge/QueryEnginePrototype>.

A.1. Plugin architecture

A.1.1. Plugin Development and deployment

Leads Plugins are POJOs (Plain Old Java Objects) implemented by platform users. As a result, users can use any third-party library of their preference or need. These dependencies must be packaged in the plugin jar and uploaded to the system using the Java client library.

Each plugin has the following attributes:

1. An **ID** that must be unique to store the plugin in the System Plugin Repository cache,
2. A **Jar** which contains the plugin code and its library dependencies,
3. A **XML configuration** file used by the user to parameterize the plugin on deployment,
4. A **Class name**, which is used to instantiate the plugin.

These attributes are packaged in a PluginPackage class. The LEADS Query Engine uses a KVS Cache (System Plugin Repository) to store all plugins. When a user deploys a plugin to a target table, the plugin is added to another cache (Active Plugin Repository), which is used for monitoring the deployed plugins, as well as, metadata regarding the deployment of the plugins.

The user must define apart from the target table, the events she is interested in to listen from that table (create, remove, modify).

A LEADS plugin must implement the interface found in <http://www.leads-project.eu/knowledge/Plugins>. The most important functions are described below:

getId: This function returns the ID of the plugin, the id must be unique as it is used as a key to store the plugin to the System Plugin Repository.

getClassName: Returns the class name of the plugin. This function is used to instantiate the plugin.

getConfiguration: Returns the Configuration of a plugin.

setConfiguration: Sets the configuration of the plugin. We use this function to parameterize the plugin on deployment.

initialize: This function is called once during the deployment of the plugin. The plugin developer must initialize all structures in this function.

cleanup: The cleanup function is run when the plugin is undeployed, for cleaning-up the structures created during the **initialize function**.

modified: The modified method is used whenever there is an update of a tuple in a table.

created: The created method is invoked whenever a new tuple is inserted in a table.

removed: This method is called when a tuple is deleted from a table.

Plugin Upload/Deployment

A user can upload his/her plugin using the following method from the WebService client library:

```
uploadPlugin(PluginPackage plugin)
```

To deploy an uploaded plugin users invoke the following method:

```
deployPlugin(String pluginId, XMLConfiguration config, String cacheName,  
EventTypes[] events)
```

Further technical details and code examples for plugins can be found in <http://www.leads-project.eu/knowledge/Plugins>. Additionally, the template plugin project that serves as a starting point for developing new plugins with some already implemented plugins can be found in <https://github.com/leads-project/leads-query-processor-plugins>.

A.2. Privacy-Preserving Point Queries

For the requirements of LEADS, we implemented the T-Set construction of the Single-Keyword SSE Scheme proposed in [CCJ+13]. Briefly, the LEADS-adapted construction consists of an encrypted database that contains the data that needs to be protected, and an encrypted index that makes it possible to execute fast queries on protected data. Both structures are stored in the LEADS platform, and accessed in a location-transparent way.

The T-Set Implementation consists of 3 basic algorithms, `TSetSetup`, `TSetGetTag` and `TSetRetrieve`. The `TSetSetup` algorithm receives as input a collection of data and produces a secure index, which is uploaded to the cloud provider. The `TSetGetTag` algorithm receives as input the queried value and produces a token which is also sent to the cloud providers. The `TSetRetrieve` algorithm receives as input a token, traverses the encrypted index, and returns the encrypted identifiers to the user for decryption. Further details about the algorithm's implementation are described in [CJJ+13].

Our implementation of the above algorithms is separated in two parts, the client-side and the server-side implementation. In the client-side implementation (class `ClientSide`), we implement algorithms `TSetSetup()` and `TSetGetTag()`, which have access to the unencrypted-plaintext data. The server-side implementation (class `ServerSide`) handles only encrypted data, i.e., it includes only the algorithm `TSetRetrieve()`.

In the following, we present instructions that could be used by a LEADS user to employ privacy-preserving point querying.

A.2.1. Quick Start

Step 1: Set values `sk_filename`, `k`, `Svalue`

Attribute `sk_filename` specifies the path of a location on the client side, where we can securely store the encryption keys that were produced during the Setup Phase. Parameters `k` and `Svalue` are used by the algorithm's internals and are set based on the expected maximum size of the data set, and the desired querying performance guarantees. Intuitively, `k` and `Svalue` affect the index size and index performance respectively. For encrypting a data set of as much as $N=2^{30}$ tuples, we recommend setting `Svalue=500` and `k=1.5`. For handling larger data sets, one can refer to [CJJ+13] for further details on selecting the two parameters.

We use the following command to initialize the `ClientSide` class:

```
ClientSide client = new ClientSide(int Svalue, double k, String sk_filename);
```

Step 2: Preprocessing Phase (Creating the Encrypted Index and the encrypted Database (EDB))

If the input data is stored in a database follow **Step 2a**. If it is stored in a text file, follow **Step 2b**.

Step2a: Call the `client.Setup()` function that receives as inputs the appropriate database connection parameter, the name of the table to encrypt and the name of the column on which we want to perform privacy preserving point queries.

```
CStore store = client.Setup(Connection conn, String tableName, String columnName);
```

Step2b: Call the `client.Setup()` that receives as inputs the name of the input file and the number of columns on which you desire to execute Privacy Preserving Point Queries. Note that this function assumes that the column separator delimiter is ",".

```
CStore store = client.Setup(String filename, int column_index)
```

At this point, we have created the encrypted database and the secure Index. Both structures are uploaded in the LEADS platform. The following steps present how to perform queries on them.

Privacy Preserving Point Queries:

Step 1: Initialize the ServerSide class

We have to initialize the ServerSide class with the use of the following command:

```
ServerSide server = new ServerSide();
```

Step 2: Initialize the ClientSide class

We have to initialize the ClientSide class by giving as input the file location of the stored secret keys (same path location as in setup phase)

```
ClientSide client = new ClientSide(sk_fileName);
```

Step 3: Produce the encrypted query (the appropriate token)

The following command receives the query as input (e.g., a word, a name, or a salary), and outputs the token that will be sent to the server.

```
String token = client.TSetGetTag(queried_value);
```

Step 4: Send the token to the server, and retrieve the results

```
HashMap<String, ArrayList<Etuple>> eResults = server.TSetRetrieve(store, token);
```

Step 5: Decrypt the answer

```
HashMap<String, <String>> unResults = client.Decrypt_Answer(eResults);
```

Notice that both encryption and decryption always take place at the data owner's machine, i.e., the data never leaves the owner's machine in plaintext, and is never gets decrypted while being inside the LEADS platform.

To further assist LEADS developers by providing reusable code, in <http://www.leads-project.eu/knowledge/PrivacyPreservingQueries> we further present two complete examples for using the privacy-preserving point queries functionality. The first example assumes that the private data is saved in a text file, and the second imports the data from a relational database.

A.3. Mashup-based user interface description

This section documents the implemented connectors and shows how these can be applied to construct a complex workflow. To avoid repetition we describe in detail the process of configuring and executing the Read Connector, while for the rest of the connectors we only provide graphical examples.

Read Connector (Figure 5): In the case of the Read Connector, we first drag-and-drop the Read operator from the function area to the main working space. Then, we right click on the connector and choose the Configure option, in order to display the existing resources, e.g., tables Entities and Webpages. We select one of these resources and click on the Next button. At this point we are able to see the attributes of the selected resource. By clicking on Finish we complete the configuration of the Read Connector.

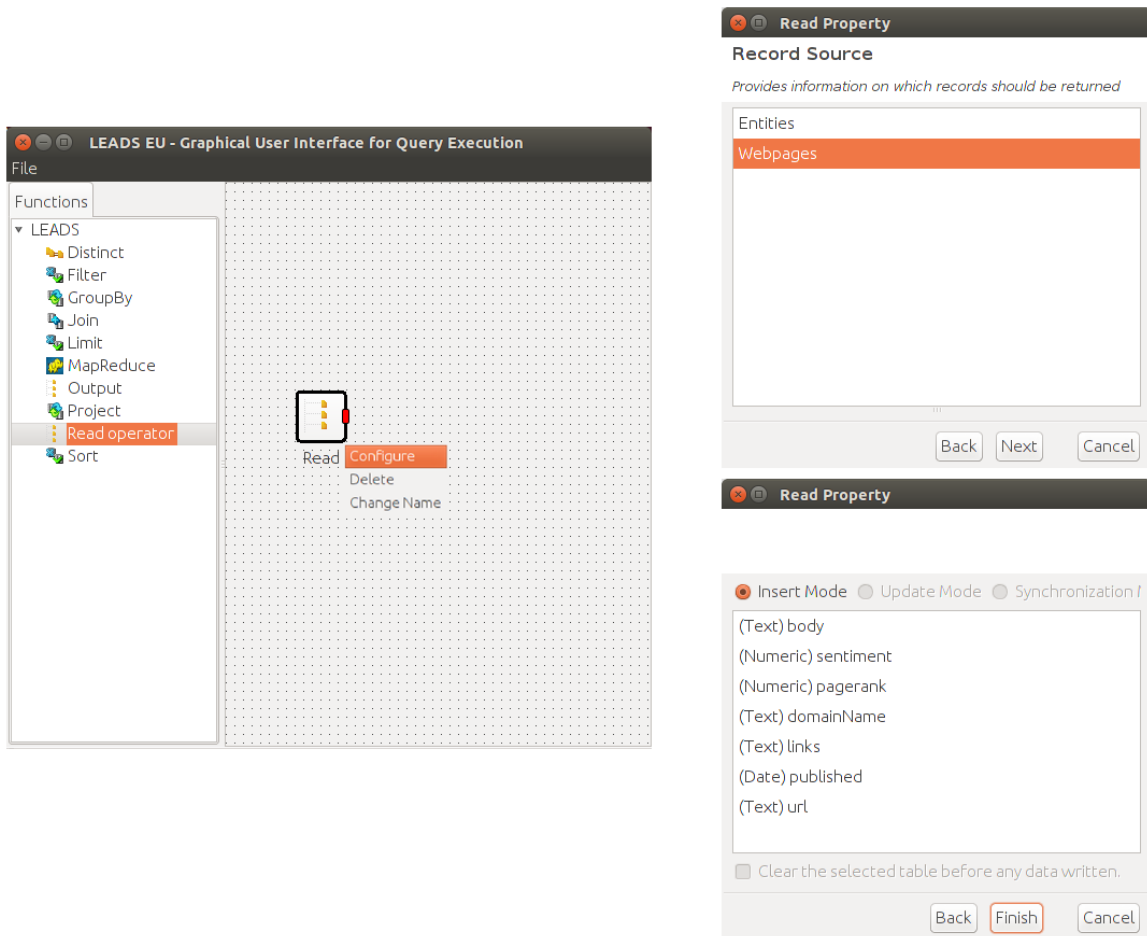


Figure 5. Read Connector

Output Connector (Figure 6): The configuration of the Output Connector requires choosing the name of the cache where the output data will be stored, and whether an existing cache could be overwritten with new data.

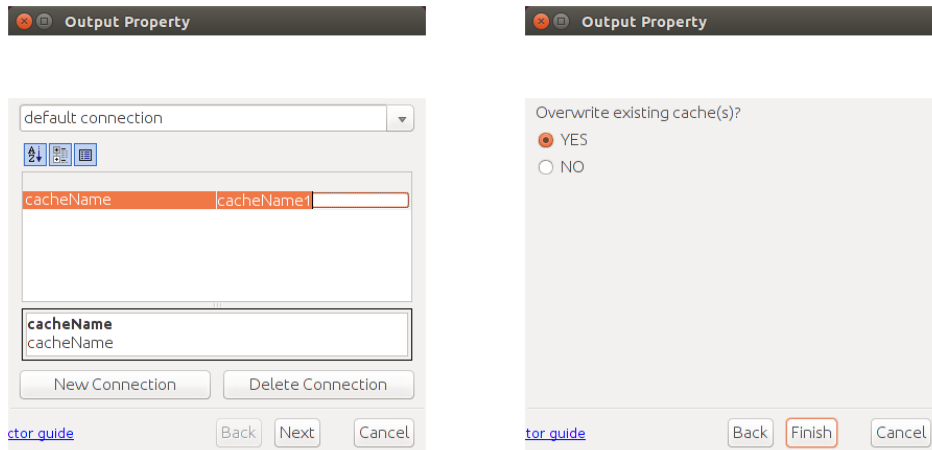


Figure 6. Output Connector

Filter Connector (Figure 7): The configuration of the Filter Connector displays the Filtration Window. The Filtration Window consists of 3 main regions ("A", "B", "C"). Region "A" comprises the input attributes, which in our example are the input attributes of table webpages. Region "B" is the working space and region "C" contains the functions of the Filter Connector. These functions are defined by numerous sub-connectors. In this example, we decided to filter the input data by keeping only the pages with pagerank equal to a value, e.g., 5. In order to do so, we drag and drop the pagerank attribute from region "A" to region "B", as well as connectors Equal To and Numeric Constant from region "C" to region "B". Then, we wire them together as shown in Figure 4 and right click on the numeric constant to select a value. Finally we select value "5" and click on the Ok button.

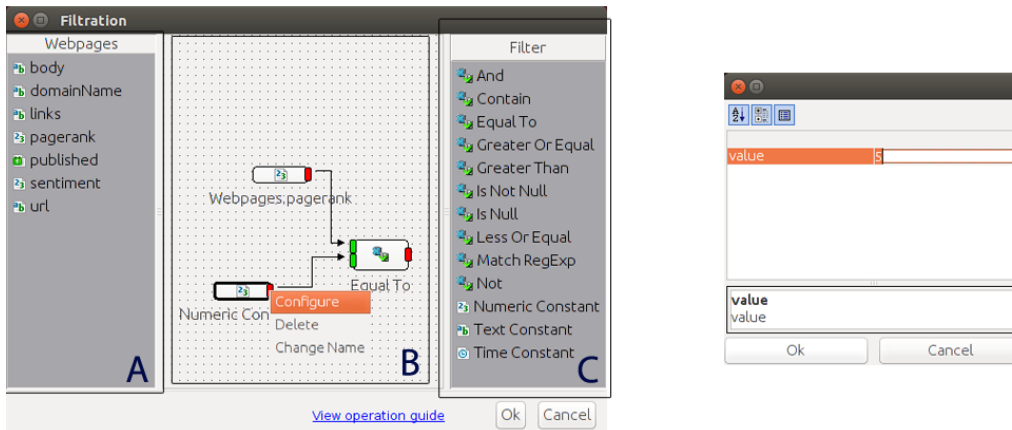


Figure 7. Filter Connector

Sort Connector (Figure 8): The configuration window of the Sort Connector is divided in the same 3 regions as with the Filter Connector, but now region "C" contains functions that correspond to the Sort Connector. Figure 8 presents two different configurations of a Sort Connector. The left window depicts sorting on the pagerank attribute values in ascending order, while the right window presents multiple sorting. Specifically, we first sort on the pagerank attribute in ascending order and the on url attribute in descending order. Then, we sort on the domainName attribute in descending order. As shown in Figure 8, we achieve this multiple sorting with the use of the MultipleSort connector.

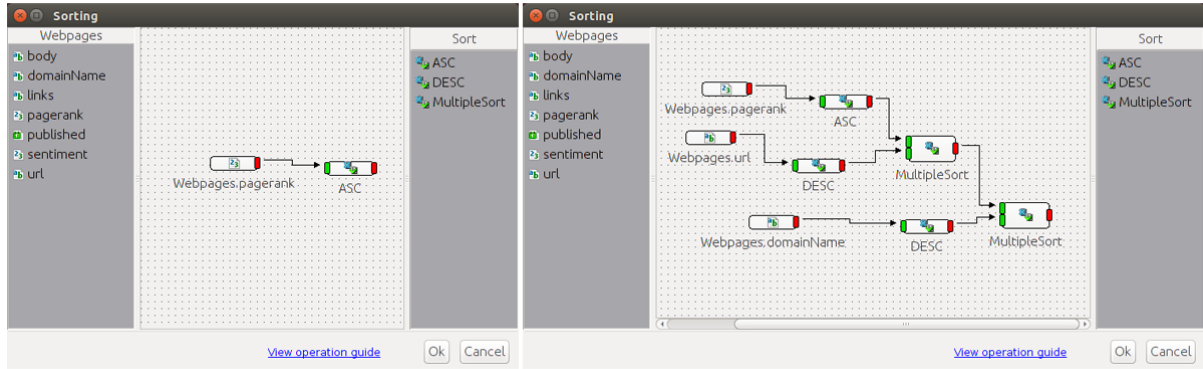


Figure 8. Sort Connector

Group By Connector (Figure 9, Figure 10): The Group By Connector window contains the Inputs and Functions tabs shown in Figure 9. The first allows us to observe the input attributes and the latter displays the supported aggregate functions. In Figure 9 we also present how to pose the following SQL query:

```
SELECT avg(Pagerank), domainName FROM Webpages GROUP BY domainName;
```

First, we drag and drop into the working space the domainName attribute (Group By attribute) and the pagerank attribute (on which we want to compute the average of the specific grouping). We wire the two sub-connectors together and click on the Edit Output button. The Edit Schema window appears on the screen (Figure 10).

Then, we click on the Add from Inputs button and we see all the input attributes. We remove them all, except from the domainName attribute (Group By attribute). Furthermore, we add a new attribute named avgPagerank. By clicking on the Ok button we return to the Group By window. At this point, we have to drag and drop into the working space the newly created avgPagerank attribute in order to wire it with the output of the Avg sub-connector. Finally, we click on the OK button to complete the configuration of the specific Group By.

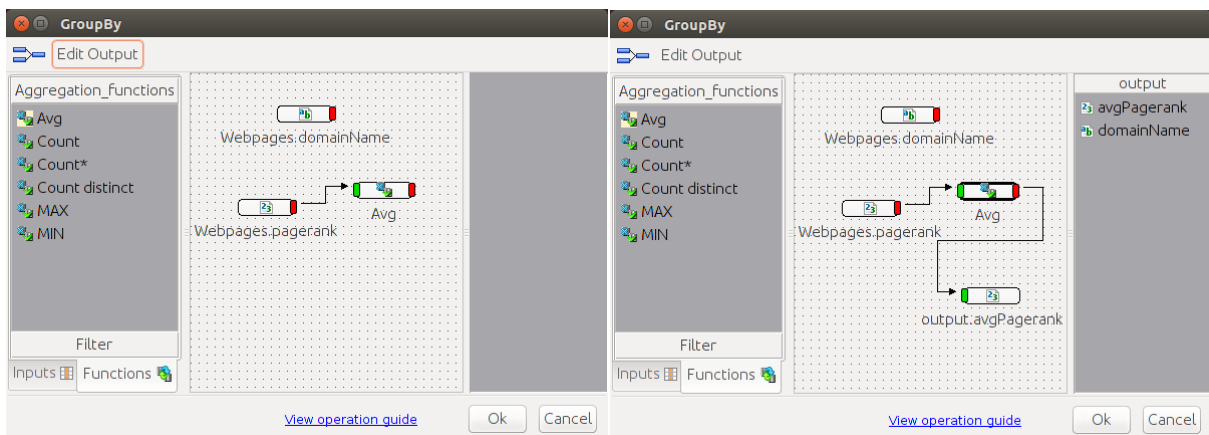


Figure 9. Group By Connector

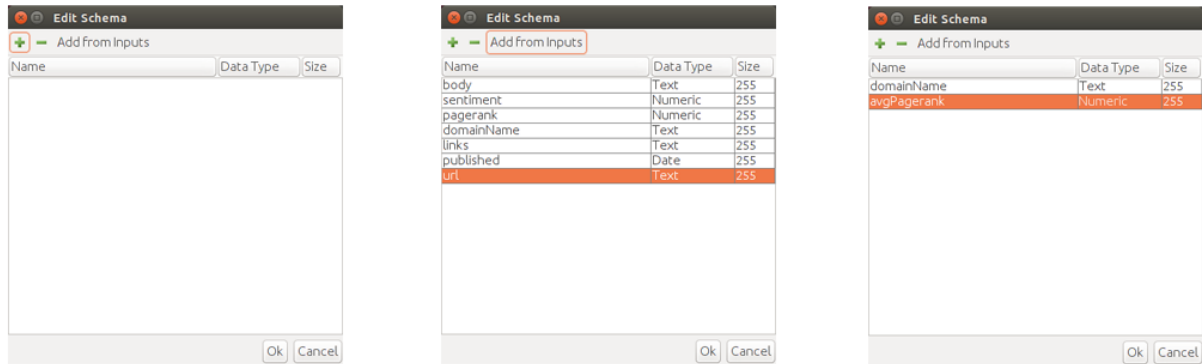


Figure 10. Editing the Group By connector

Limit Connector (Figure 11): Configuring the Limit Connector requires choosing a limit value e.g. “5” and then clicking OK.

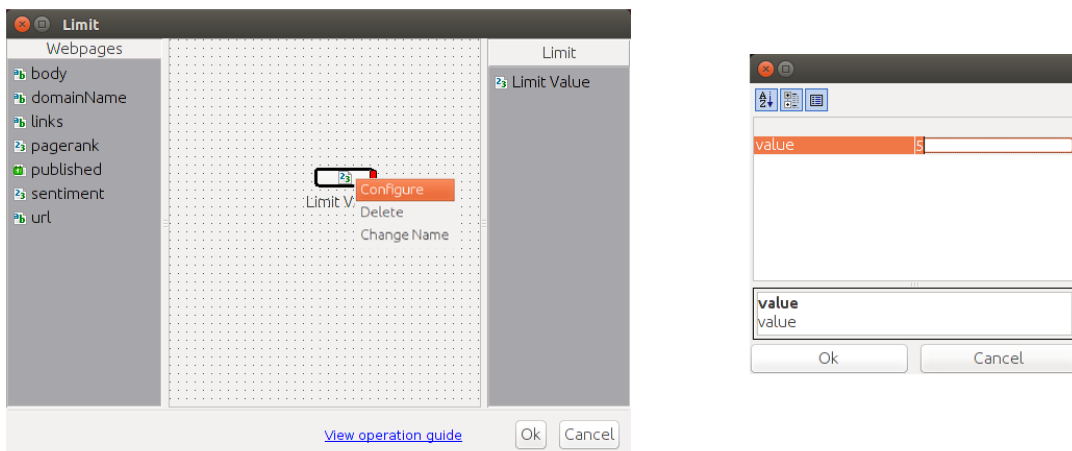


Figure 11. Limit Connector

Project Connector (Figure 12): The following figure presents the Projection Window before editing the output data and after. The functionality of the Edit Output button in this case is the same as in the Group By connector. In this example we implement the following query:

```
SELECT body, links, pagerank, published FROM Webpages;
```

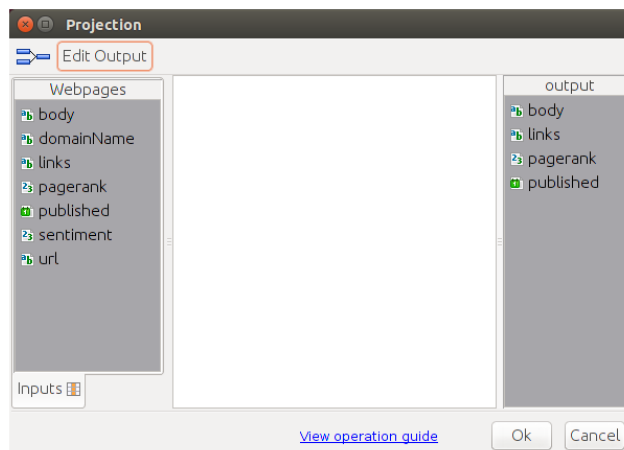


Figure 12. Project Connector

Distinct Connector (Figure 13): The following figure depicts how to choose the distinct values of the domainName attribute.

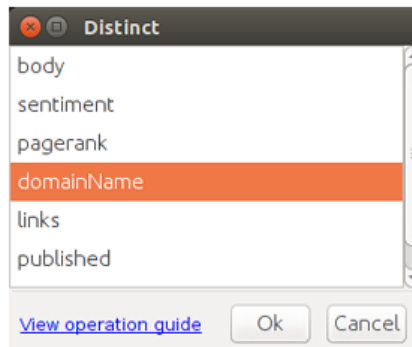


Figure 13. Distinct Connector

Equi-Join Connector (Figure 14): It is necessary to configure and wire two Read Connectors prior to the Equi-Join Connector. In our example these Read Connectors are based on the Webpages and the Entities tables. After wiring the two connectors we right click on the Join connector and select the Configuration Option. Then, we click on button "+" and select the attribute of each table on which we want to perform the Equi-Join function.

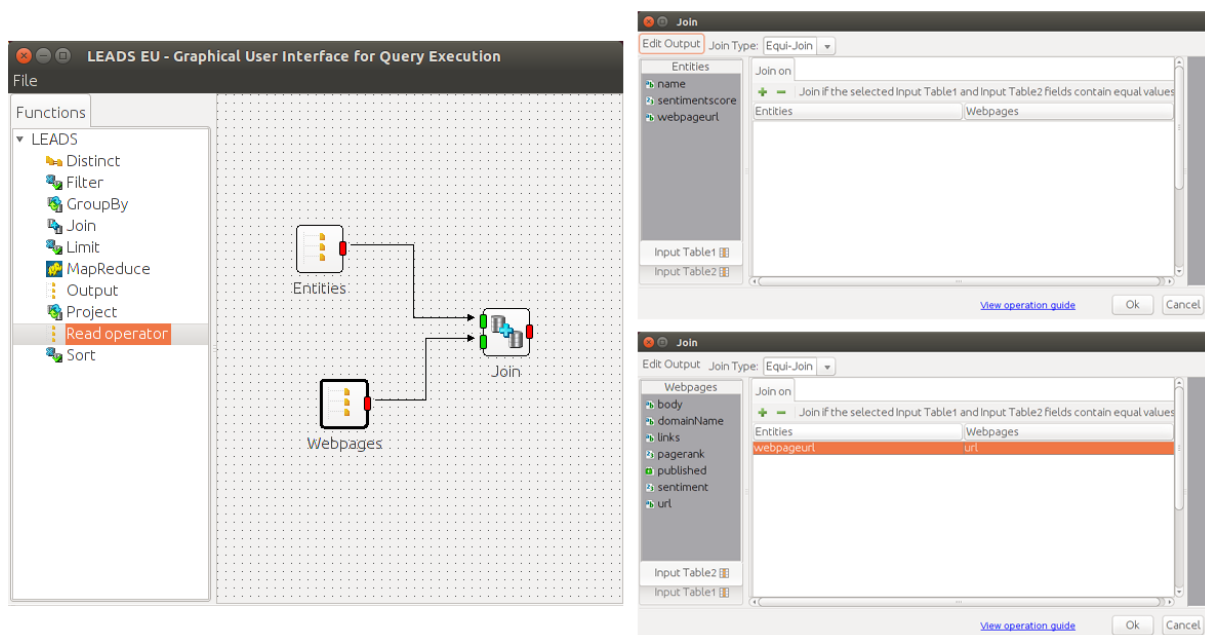


Figure 14. Equi-Join Connector

MapReduce Connector (Figure 15): For the MapReduce connector we first need to specify 2 file locations, the Configuration Path and the Jar Path. The configuration window of these two sub-connectors is illustrated in the right upper image. Then, we have to define 2 string values for the Mapper and the Reducer. The configuration window of these two sub-connectors is presented in the right lower image.

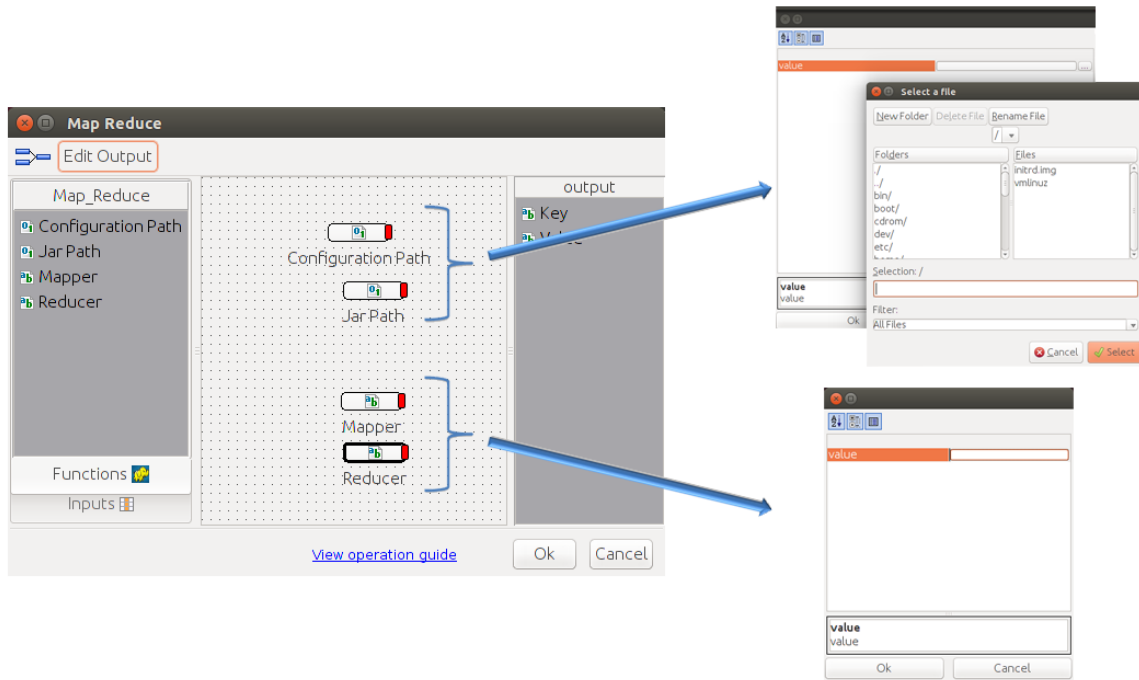


Figure 15. MapReduce Connector