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:	D4.4
Title of Deliverable:	Fully-featured prototype of scheduling and data placement supporting agile elasticity and cost-aware and energy-aware scheduling
Contractual Date of Delivery:	M36 – 9/30/2015
Actual Date of Delivery:	9/28/2015

Abstract

In this deliverable, we present our final prototype for data placement, and cost- and energy aware scheduling. The deliverable highlights the achievements we made within the final year with regards to data placement, scheduling of processing and crawling tasks in micro-cloud environments such as the LEADS platform.

We first present our approach for distributing data across the micro-cloud infrastructure of LEADS, which is well suited for the targeted the Data as a Service paradigm. Our evaluation shows that, with a high number of distributed micro-clouds, data locality can be considerably improved.

With regards to scheduling, we extended our initial prototype with artificial neural networks, which we train using genetic algorithms. We were able to consider several additional metrics for scheduling, such as data transfer and computation costs, and energy consumption independently, and to perform a scheduling that considers all those dimensions synergistically. Our evaluation shows that it is possible to considerably decrease overall costs using our approach when compared to a simple random scheduling.

List of Contributors

Name	Organization	E-mail
Wojciech Barczynski	Cloud & Heat	wojciech.barczynski@cloudandheat.com
Marcel Gädig	Cloud & Heat	marcel.gaedigk@cloudandheat.com
Daniel Poelzleithner	Cloud & Heat	daniel.poelzleithner@cloudandheat.com
Steve Schmerler	Cloud & Heat	steve.schmerler@cloudandheat.com
Jens Struckmeier	Cloud & Heat	jens.struckmeier@cloudandheat.com
Gaurav Singh	BM-Y!	gauravonline20@gmail.com
Ioanna Tsalouchidou	BM-Y!	ioanna@yahoo-inc.com
Janette Lehmann	BM-Y!	janettel@yahoo-inc.com
Necati Bora Edizel	BM-Y!	edizel@yahoo-inc.com
David Garcia Soriano	BM-Y!	davidgs@yahoo-inc.com
Aslay Cidgem	BM-Y!	aslayci@yahoo-inc.com
B. Barla Cambazoglu	BM-Y!	barla@yahoo-inc.com
Hossein Vahabi	BM-Y!	puya@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
Aggelos Aggelidakis	TSI	aaggelidakis@softnet.tuc.gr
Eleftherios Chatzilaris	TSI	echatzilaris@softnet.tuc.gr
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@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
Jons-Tobias Wamhoff	TUD	jons@inf.tu-dresden.de
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Raluca Halalai	UniNE	Raluca.Halalai@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Valerio Schiavoni	UniNE	Valerio.Schiavoni@unine.ch
Anita Sobe	UniNE	Anita.Sobe@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch

Document Approval

	Name	Email	Date
Approved by WP Leader	Christof Fetzer	christof.fetzer@tu-dresden.de	2015-09-15
Approved by GA Member 1	Hossein Vahabi	puya@yahoo-inc.com	2015-09-27
Approved by GA Member 2	Minos Garofalakis	minos@softnet.tuc.gr	2015-09-25

Contents

List of Figures

1. Executive Summary

In this deliverable, we describe our final prototype of the data placement and scheduling component developed within WP4 of the LEADS project. The component has the following two objectives: First, the partitioning and placement of data which is continuously crawled by the LEADS platform; and second, the scheduling of data processing and crawling tasks across the infrastructure.

In the first part of this deliverable, we describe our approach for placing data retrieved through a continuous crawl on a set of available micro-clouds which comprises the LEADS platform. In order to do so, we follow a *static partitioning* approach, i.e., decide upfront how to split the data and where to place those splits. The upfront partitioning and placement approach requires a sample crawl, which ideally provides a good approximation of the coming data of the following continuous crawl of the platform.
During the third year, we analysed the publicly available CommonCrawl (Common Crawl, 2015) dataset, which comprises roughly 154TB of raw data and serves as a sample crawl for exercising our algorithms to perform data partitioning and placement. As shown in our evaluation, we can improve data locality, i.e., minimize the distance between micro-clouds and servers, if splits are chosen sufficiently small.

In the second part of this deliverable, we present our approach for scheduling of processing tasks with regards to energy efficiency and cost awareness. The objective of the scheduler component is to place data processing tasks close to the data sources in order to save network bandwidth, prevent costly data transfers and taking both, costs and energy consumption jointly into account.
During the past 12 months, we extended our previously presented approach by utilizing artificial neural networks. Using genetic algorithms and commonly used queries provided by the AMPLab benchmark (AMPLap, 2015), our approach allows the training of neural networks for future scheduling decisions. We compared our approach with a simple random scheduler and were able to achieve considerable cost savings. Using genetic algorithms for the training of the neural networks allowed us to reach effective results within a reasonable time frame.

2. Data Placement

2.1 Introduction

In this section, we present our approach for placing data across a set of micro-clouds in the context of the LEADS project.

The LEADS platform targets the storage of the following two types of data: 1) ***websites*** which are continuously crawled using the distributed crawler developed within WP1, and 2) ***query results*** originating from processing tasks of the query engine developed within WP3.

We assume that the web corpus will make up the larger fraction of the data set, since a reduction of the data is generally performed as a first step when processing data of such kind. Data reduction can be achieved, for instance, through the extraction of relevant fields from metadata such as URLs pointing to other websites in order to build a distributed web graph. Another example for data reduction involves the usage of simple filtering techniques such as considering only sites that match a certain criteria like the origin or a language identifier for a follow up sentiment analysis.

Because of its enormous storage volume, the web corpus cannot be placed on a single micro-cloud. Two problems have to be solved: first, the corpus needs to be partitioned into equally sized chunks/buckets, and second, the buckets must then be placed across the set of available micro-clouds. In order to increase the availability of the data, it is furthermore beneficial to place replicas of the previously created buckets on different regional zones in order to prevent unavailability of data due to downtime or maintenance of micro-clouds in a certain zone.

Data partitioning and placement can be either performed at ***design-time***, i.e., prior to the actual crawl or at ***runtime***, i.e., while the data is being collected. In the first case, the partition boundaries must be defined upfront which requires to run a sample crawl in order to determine appropriate bucket sizes and a placement, while in the latter case, a simple hashing mechanism utilizing the URL as key can be used. Although this approach avoids the execution of a costly sample crawl, it will lead to a data placement that does not take data locality into account, imposing unnecessary remote reads and data transfers when querying and processing the data later on.

Consider as an example the following three domain names: www.abc.com, www.abc.com/news and www.xyz.de. Using a simple hash would place for instance the first and last domain on a micro-cloud A while the second one would end up on a different micro-cloud B although the first and second domain share the same domain, and hence relate to each other. Design-time data partitioning and placement can therefore improve data locality with regards to related content.

2.2 Approach

For LEADS, we chose the design-time data partitioning approach, to achieve a better uniform distribution of data across micro-clouds as well as a co-location of content-wise related data. As mentioned in the introduction, a sample crawl is required in order to determine bucket boundaries for data partitioning. Once the boundaries are defined, buckets must be placed across micro-clouds. For the data partitioning and placement of buckets, we have the following two objectives:

1. Related content shall be *packed/stored* into the same buckets
2. Buckets shall be placed close to the location the data has been crawled from (i.e., the data source)

The fulfillment of the first requirement ensures that related content resides on the same micro-cloud, decreasing the amount of remote-reads and bandwidth usage of inter-micro-cloud links. The fulfillment of the second requirement ensures that content is always stored in close proximity to its providers and potential consumers.

Consider for example a site such as **www.spiegel.de**, a German news content provider: since the site is hosted in Germany as well as its content primarily consumed locally, it is beneficial to choose a micro-cloud located in the same region for storing the site's content. Moreover, there is also a high probability that consumers in Germany will select the site's content for query processing leading to an overall better data locality for the query processor developed within WP3.

2.2.1 Data Partitioning

2.2.1.1 Bucket Boundaries

To achieve a co-location of related content when splitting up the web corpus, we first reduce domain names that have more than three levels by removing all remaining parts from the original URL as used when crawling a site. For example, consider an URL like **www.leads-project.eu/technology**. Using the reduction step mentioned above, this particular URL will be mapped to a key solely named **www.leads-project.eu**. The mapping has the advantage that all subpages to this site will be mapped to the same key, hence co-located on the same bucket and therefore stored at the same micro-cloud in the end.

After the first mapping step, we furthermore convert the key to its reverse domain name. Hence, in the given example, **www.leads-project.eu** will become **eu.leads-project.www** now.
Using the reverse domain names, we will perform a ***lexical sort*** on the set of those keys next. The sorting serves two purposes: First, data items with a similar prefix, i.e., identical top level domain name e.g. eu.* will automatically turn into neighbors using this sorted key-set, and second, a simple relation can be used in order to determine the appropriate bucket for a newly discovered domain.

eu.leads-project.www
…
eu.new-project.www
…
eu.paradim-project.www
…

As an example, consider the set of buckets shown above. As depicted in the table, the domain space has been partitioned into two buckets: One that starts with "**eu.leads-project.www**" as key and another one with "**eu.paradim-project.www**". Since the sample crawl does not comprise the complete set of the sites available in the web, new sites may be discovered at any time during the continuous crawl such as "**eu.new-project.www**". Since URLs are ordered lexicographically, the newly discovered URL can be easily assigned to the existing bucket starting with "**eu.leads-project.www**" as key through a lexical comparison of the previously defined bucket boundaries.

2.2.1.2 Bucket Size

Forming buckets with an appropriate size can be achieved in two different ways: First, the number of collected URLs can be used in order to determine appropriate boundaries what we will refer as ***count-based*** approach. The second approach is based on the accumulated number of Bytes which we will call the ***size-based*** approach.

For both variants the total number of URLs and total size of the sample crawl must be determined as a first step. In a second step, the complete URL space can then be traversed from top to bottom using an accumulation counter in order to create the desired number of splits where each split/bucket comprises an equal number of URLs or Bytes.

The threshold for the traversal determines the size of each bucket and the total number of buckets in the end. A common approach is to decide on the number of desired buckets first and then to compute the bucket size as follows:

$$size_{bucket} = \frac{size_{total}}{number_{buckets}}$$

As a rule of thumb, the number of buckets should be chosen sufficiently large as it allows a fine-grained placement of the previously formed buckets, maximizing co-location and data locality since each bucket comprises only a few domains which can be placed as close as possible to the location where they have been crawled from. However, choosing too many buckets may lead to a performance decrease, as the bucket boundaries must be kept in memory for fast lookups during crawling and query processing. Hence, there is a trade-off between a fine-grained data partitioning with good locality and lookup performance.

2.2.2 Bucket Placement/Assignment

After the initial creation, the buckets must be assigned to the set of available micro-clouds. The objective of the assignment is to place buckets as close as possible to the region of the data source.

Since a bucket comprises several URLs that often originate from several distant locations, we first compute the centroid (i.e., center of gravity) for each bucket[1]. Assuming equal masses for each point:

$$\{lat, lon\} = \left\{\frac{\sum_{i=1}^{n} lat_i}{n}, \frac{\sum_{i=1}^{n} lon_i}{n}\right\}$$

Next, we rank all available micro-clouds by their distance to the center of gravity of each bucket. After the ranking, we iteratively assign buckets to their closest micro-cloud if space is still available, or else it is assigned to the second, thirds etc. closest one if the chosen micro-cloud does not provide sufficient capacity to hold the data.

2.3 Implementation

We choose the Java programming language for the implementation of the previously described mechanisms and algorithms for data placement in LEADS. The choice has the advantage that the ***partitioner*** object, responsible for assigning newly discovered URLs to the existing buckets and for performing lookups, can be directly integrated into Infinispan as well the Nutch-derived Crawler developed within WP1. Note that the ***partitioner object*** holds a tree in memory with the previously determined bucket boundaries to efficiently perform lookups to determine what bucket a certain URL belongs to.

In order to compute the bucket boundaries, a sample crawl is required to run. Instead of running a sample crawl ourselves, we used the publicly available CommonCrawl data set (Common Crawl, 2015). The data set comprises around over 145TB in size and holds more than 1.81 billion webpages. Although the data set spans only a fraction of the whole web, it can be considered as a representative part of the web to provide a good estimate of where the majority of sites and content is hosted. The crawl we used for the evaluation was crawled in July 2015.

For the deployment and location of micro-clouds, we used two different sets of micro-clouds for the evaluation of the assignment of buckets to micro-clouds as we will describe more in detail in the following section. The first dataset comprises all Google datacentres, i.e., 12 in total while the second set comprises the list of officially registered datacentres at www.datacenter9.com, which holds currently over 3k datacentres across more than 40 countries. We believe the second dataset is a representative candidate for the future deployment of micro-clouds run by our project partner Cloud&Heat. For our evaluation we used both datasets.

In order to determine bucket boundaries and assign those buckets to micro-clouds, we utilize MapReduce (Ghemawat, 2004). The MapReduce paradigm originates from functional programming, however, has been also successfully applied to big data processing as described by authors in (Ghemawat, 2004). MapReduce is characterized by essentially two processing stages: A map phase which is usually used for filtering and transformation of data, and a reduce phase where all data tuples sharing the same key are passed to a stateless reduce function. More details on the paradigm can also be found in the respective deliverables of WP3.

We expressed our data transformation tasks to form and assign buckets to micro-clouds as four consecutive MapReduce jobs/stages, each one detailed in the reminder of this section.

[1] Note that we retrieve the geolocation for each URL through the freely available GeoIP database.

2.3.1 Stage 1 – Data Extraction, Enrichment and Aggregation on Reverse Domain Level

In the first stage, we extract the relevant metadata from the CommonCrawl (Common Crawl, 2015) data set during the mapping phase. This involves parsing the JSON format of the provided WAT files (WARC file format, 2015) in order to retrieve the size of the crawled site, its IP address and the domain name.

Prior to emitting tuples to the reduce stage of the job, we perform a data enrichment step by looking up the IP address to determine its geographical location. To this end, we use the freely available GeoIP database (GeoIP2 City Database , 2015) which provides a mapping of IP addresses to country code, city and geospatial information such as latitude and longitude. As a key for the following reduce stage, we construct the reverse domain name and truncate superfluous parameters from the query string.

In the following reduce stage, a first aggregation is being performed where several URLs sharing the same key are consolidated by aggregating their total size and number of pages, and by computing their centroid as described in the previous section.

2.3.2 Stage 2 – Bucket Formation

In this stage, the entire web corpus is split up into individual buckets. In order to achieve this, we first use an identity mapper that reads in the file produced in the previous stage. Using a single reducer, the input is concatenated using a strict ordering based on the provided key. This mechanism can now be used in the reduce step to form equally sized chunks where the size of a chunk can either be defined in terms of number of pages or accumulated bytes. The aggregation mode as well as the threshold can be set through appropriate parameters.

2.3.3 Stage 3 – Micro-cloud Ranking

In the following stage, we use an identity mapper and perform an aggregation using the reduce method by computing the centroid from the URLs in each bucket. The input data is further enriched by computing the distance from the centroid to each of the micro-clouds, and placing the distances in a sorted set representing a distance ranking. In addition to the centroid computation, the average and total distance of each URL to the centroid is computed and emitted as well.

2.3.4 Stage 4 – Micro-cloud Assignment

In the final stage, the buckets are assigned to their closest micro-clouds. The assignment is done in the reduce phase of this stage where the reduced data set from the previous stage is fetched into memory and then the buckets iteratively assigned to the micro-clouds. The assignment is done in the close() method of the reducer producing the final output.

2.4 Evaluation

In the following, we present the evaluation of our approach. For the experiments, we used the CommonCrawl data set which serves as our sample crawl comprising roughly 500 GB of raw data after pre-processing, spread over almost two billion pages and 25 million different URLs as shown in the following table:

Total amount of data set	527 GB
Total amount of pages	1,814,077,431
Total amount of domains	25,064,385

In order to evaluate our approach, we varied several parameters such as the number of buckets and their associated thresholds or the number of micro-clouds, and compared the distances between the locations data the content was retrieved from, i.e., the data source and the micro-cloud selected for storing the data. To measure the quality of the achieved placement, we target to *minimize* the average distance between micro-clouds and the data sources. We also investigate the placement of data when the capacity of a micro-cloud is bounded and for the situation when we would have theoretically unlimited storage capacity.

With regards to the evaluation of the number of micro-clouds, we used two different data sets: The first dataset represents 12 micro-clouds located in close proximity to the Google datacentres. Since Google can be considered as a global leader for various online products such as search engine and email provider, we assume a good global service coverage of those 12 datacentres. For the second dataset, we used the datacenter9.com database which consists of 1708 registered datacentres world-wide. We believe that this number of data centres matches best the future deployment of the micro-cloud infrastructure.

For a better understanding of the locality of datacentres across the globe, we visualized the two datasets using Google Maps in conjunction with Google *FusionTables*. Figure 1 shows the location of the 12 Google datacentres while Figure 2 depicts the location of datacentres registered at www.datacenters9.com.

Figure 1 Locations of Google datacentres, i.e., 12 micro-clouds.

Figure 2 Locations of datacentres registered at datacenters9.com, i.e., 1708 micro-clouds.

2.4.1 Average Distance Micro-Cloud – Content Retrieval

In our first experiment, we varied the size of a bucket: 10 MB, 20 MB and up to 100 MB. We furthermore executed the experiment twice using the 12 and the 1708 micro-clouds for an assignment in the first and the second round, respectively. For the assignment of buckets to micro-clouds, we as-

sumed unlimited storage capacity at a micro-cloud first. The outcomes of our experiment are presented in Figure 3 and Figure 4.

As shown in Figure 3, the average distance from a micro-cloud to its data source increases with the chosen bucket size. This effect is expected: with smaller buckets, a finer grained placement can be performed since buckets comprise a smaller set of URLs. In theory, a perfect placement can be achieved if a bucket comprises only a single URL: in this case, the closer micro-cloud would be chosen systematically. However, as mentioned previously, such a fine-grained placement is unpractical under the expected data volume to be crawled.

If we further compare the average distance of micro-cloud to the location the data was crawled from with respect to the number of micro-clouds used, we can observe that the distance is several orders of magnitude lower when a higher number of micro-clouds are being used. Intuitively, a larger number of micro-clouds allows better data locality, i.e. it is easier to find a micro-cloud that is closer to the data source.

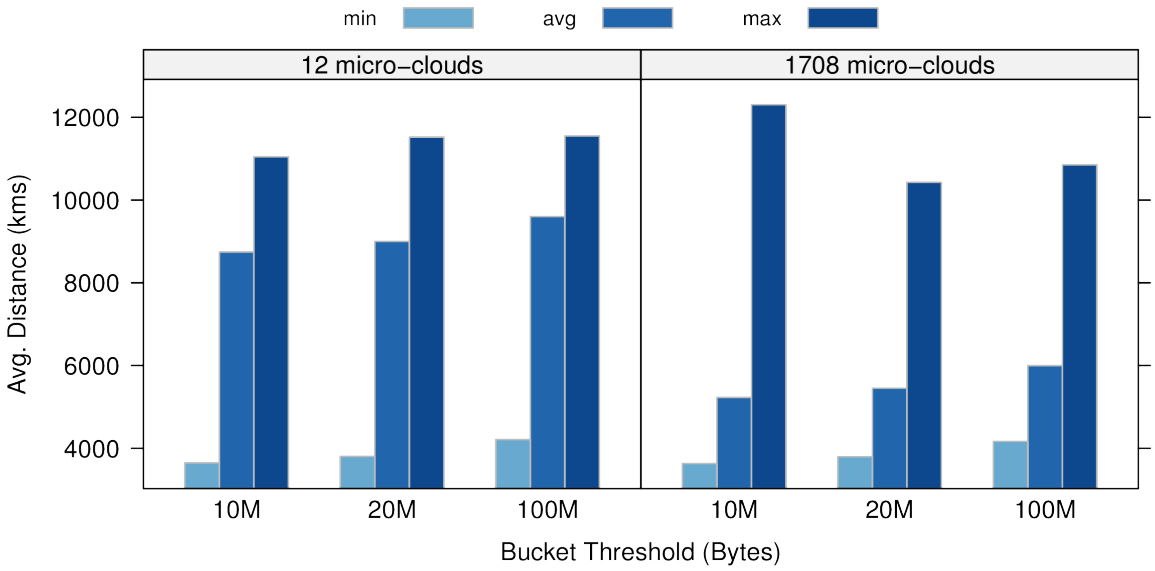

Figure 3 Distance between URL crawling origin and micro-cloud (w/o capacity limit)

Figure 4 presents the cumulative distribution function of micro-clouds to their data sources. As shown in the figure, 50% of the micro-clouds are within six and nine thousand kilometres range of their data sources, respectively. Furthermore, a higher number of micro-clouds leads to a placement with generally a lower distance between the data source of a bucket and its associated micro-cloud.

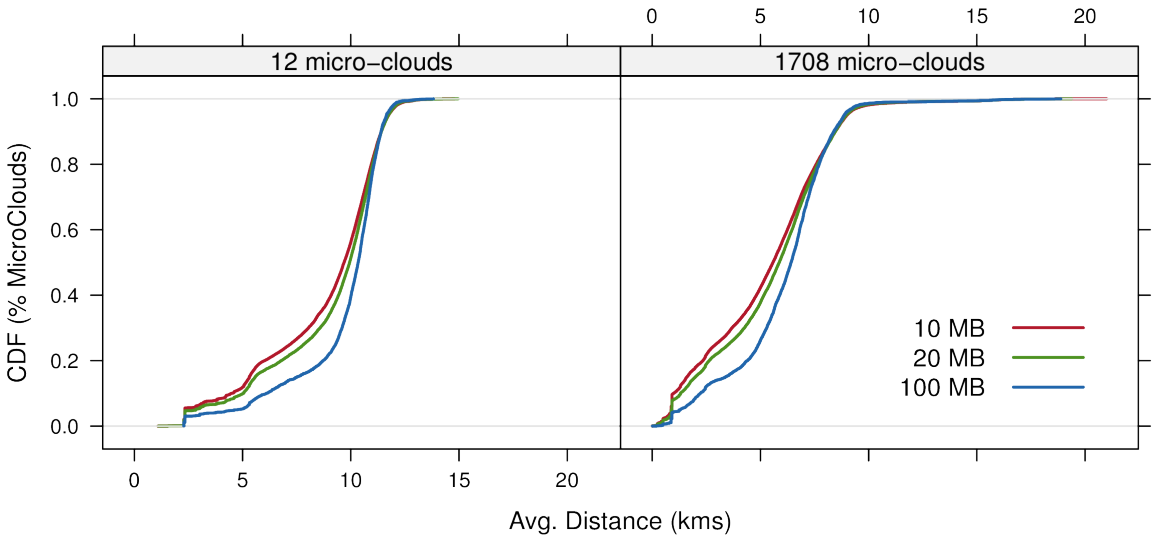

Figure 4 Distance distribution URL crawling origin and micro-cloud (w/o capacity limit)

Figure 5 shows the utilization of storage for each of the micro-clouds. As mentioned previously, the first experiment was performed under the assumption of unbounded storage capacity. As shown in the figure, such an unconstrained placement leads to a quite unbalanced utilization of micro-clouds.

For example, when using only 12 micro-clouds, a single micro-cloud has to maintain up to 80% of the data which clearly exceeds its storage capacity and leads to imbalances when processing the dataset afterwards. Also an increase of micro-clouds does neither improve nor evenly balance the assignment of buckets to micro-clouds as shown in the rightmost graph in Figure 5. We therefore re-run the experiment using our improved algorithm where the storage capacities as well as the utilization are taken into account in the bucket assignment process.

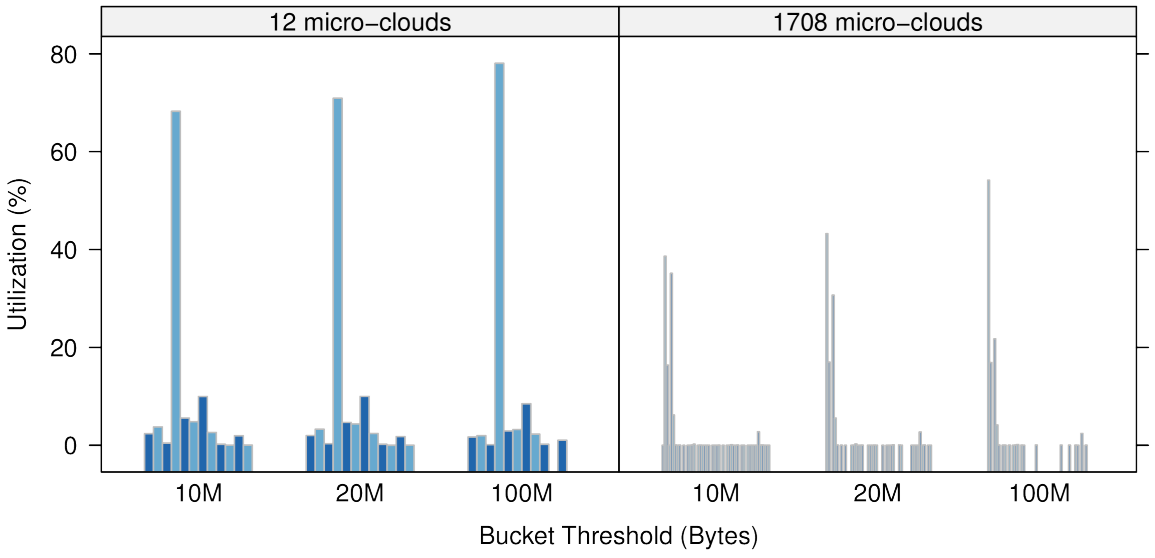

Figure 5 Storage utilization of each micro-cloud (w/o capacity limit)

2.4.2 Storage Capacity-Constrained Placement

In the following experiment, we constrain the capacity of each micro-cloud. We assume that each micro-cloud has equal storage capacity, which is a realistic assumption as the micro-clouds provided by Cloud&Heat are equipped with identical numbers of nodes and storage capacity.

The outcome of these experiments are depicted in Figure 6, Figure 7, and Figure 8. As shown in Figure 6, the distance between data sources and micro-clouds increases with bucket size. However, if we compare the average distance with regards to the different placement strategies, i.e., unconstrained vs. capacity constrained placement, we can observe that the average distance increased by approximately 60%. For the 12 micro-cloud case, the distance was originally nine thousand kilometres while with the constrained placement, we can now observe 16 thousand kilometres in average.

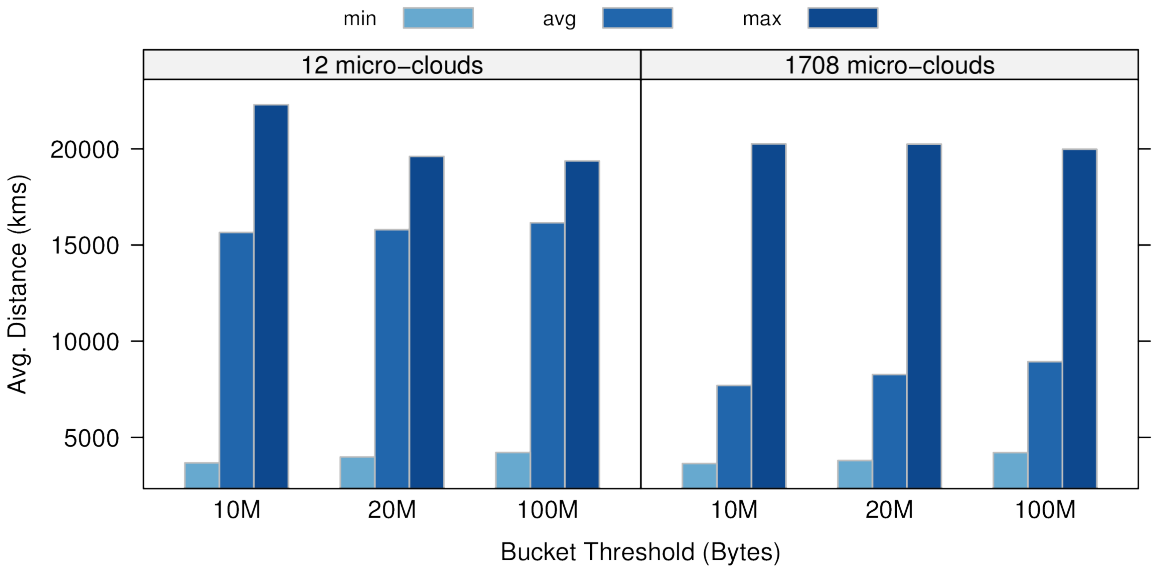

Figure 6 Distance between URL crawling origin and micro-cloud (with capacity limit)

Also with regards to the distribution function, we can identify a clear difference between those two strategies as shown in Figure 7. We can observe a longer tail for both configurations (12 and 1708 micro-clouds). The reason is that buckets must be moved further away as all micro-clouds in close proximity are fully loaded. If we compare the results of Figure 7 with those presented in Figure 4, we observe how the curves shifted slightly to the left, a result consistent with the average distances (see Figure 6).

To validate the balancing of our placement strategy, we present in Figure 8 the storage utilization for each micro-cloud. Clearly, we can see an almost equal distribution of data across the 12 and 1708 micro-clouds.

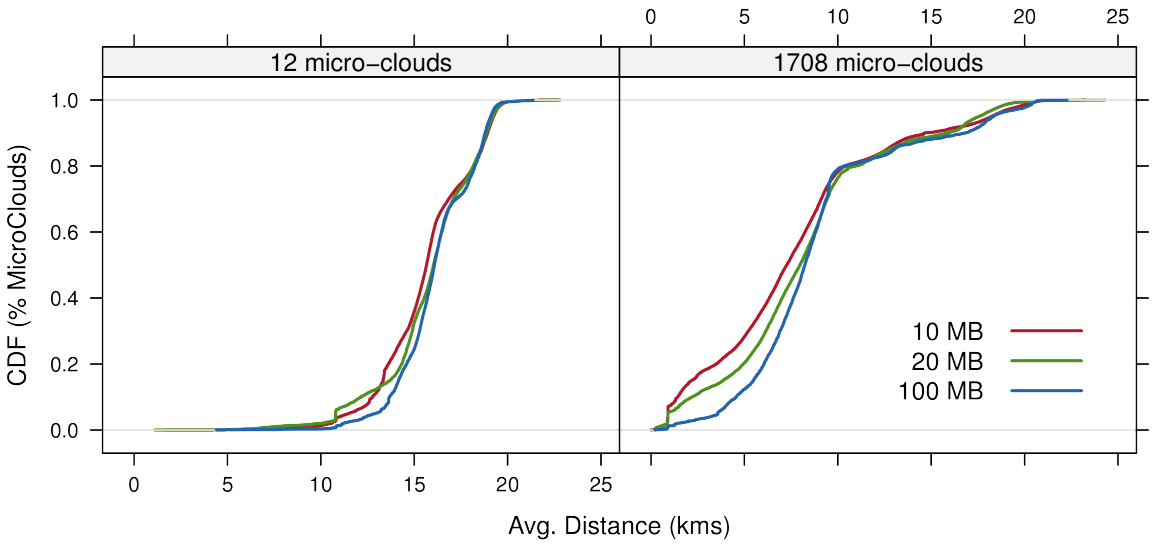

Figure 7 Distance distribution URL crawling origin and micro-cloud (w capacity limit)

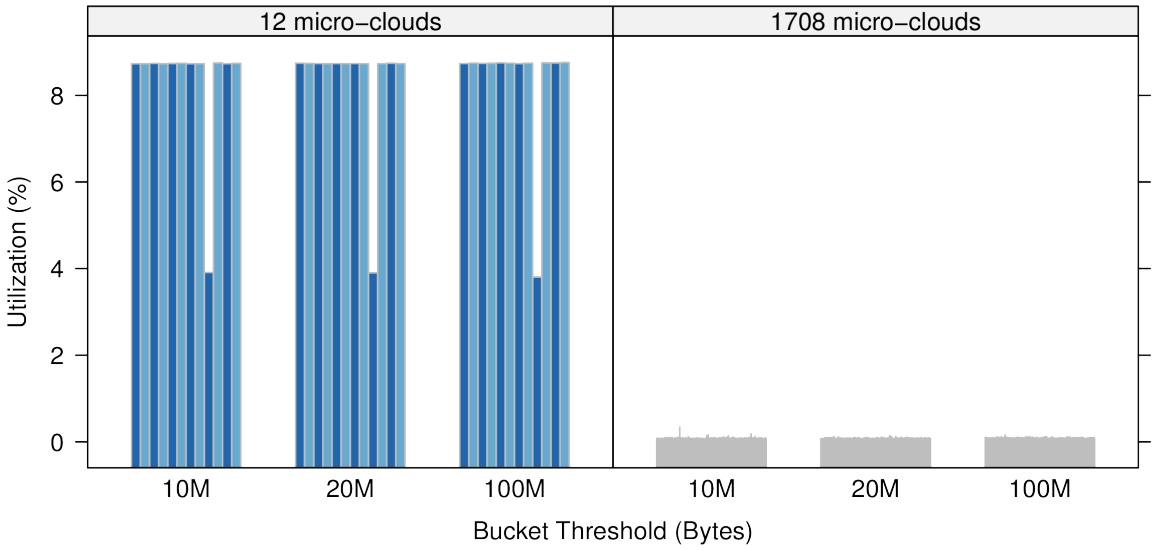

Figure 8 Storage utilization of each micro-cloud (w/ capacity limit)

2.5 Summary & Conclusion

In this section, we presented our approach for a static data placement for crawled data across a set of micro-clouds in the context of the LEADS project. We first described our general idea on how to split up the domain space using a static *partitioner*, followed by an in-depth description about our data placement/assignment across the set of available micro-clouds.

In our evaluation we showed that the distance between data sources and the micro-clouds can be minimized with an increasing amount of micro-clouds, confirming our initial assumptions that the distributed infrastructure as envisioned by the LEADS project has several advantages. We furthermore investigated the impact of the distance of data storage to the data source with respect to bucket sizes. We have seen that a higher number of buckets improves with a higher number of buckets. As dataset for the sample crawl we used the publicly available *CommonCrawl* which is a good approximation of the web.

3. Energy-aware and Cost-based Scheduling

3.1 Introduction

Queries targeting the LEADS platform can be of different size, complexity, and exposing different data access workload. To design an efficient scheduler, we provide a simulator that allows evaluating the scheduler's goodness (in terms of time and load balancing). The main role of this simulator is to predict the behavior of query executions in a simplified manner.

In the following we describe the query workflow as a basis for the *Query Cost Estimation Simulation*.

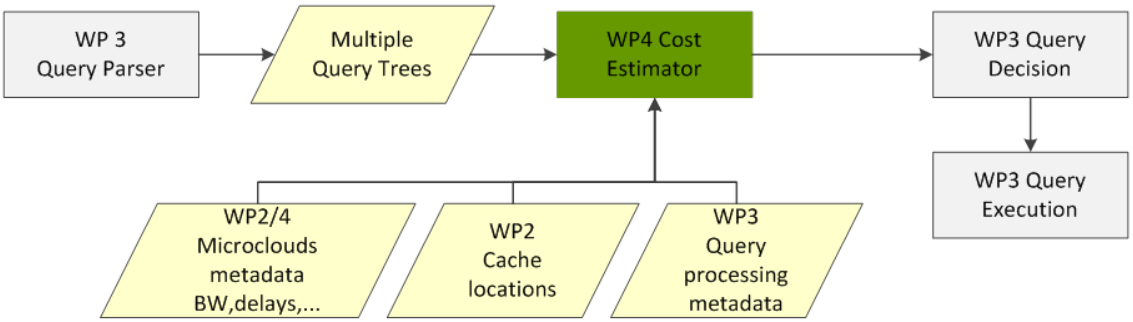

Figure 9 Overview of the LEADS query workflow

Figure 9 sketches the integration of the WP4 query estimator in the form of a simulator. The simulator collects information from WP3 that are related to queries, information on micro-clouds and on cache locations. Based on these inputs the simulation is built, and the scheduling performed.

3.2 Assumptions

1. The estimation of the costs of a query can be done based on multiple criteria, but these criteria will be executed separately. In our first version, we consider the metadata of a query execution as the starting point.

   As an example: Let us consider that we want to know the cost of a query based on duration and on energy. The cost estimator will look at each criterion in isolation. Hence, there will be two outputs for one query, one for duration, and one for energy. A combined estimation that takes into account both parameters can be optionally done by the Query Decision process of WP3.

2. The cost estimation component targets the micro-cloud-level and **not** the node-level. This assumption is based on the usage of Infinispan, which uses all nodes of a micro-cloud for a MapReduce job. As a consequence the goal is ***not*** to assign query parts to a single node, but rather selecting the correct micro-clouds for execution of the query. This still means that the query tree placement is sufficiently complex.

3. The decision on which query plan is to be executed is **not** part of the cost estimator, however can be part of the simulation.

4. The inputs to the simulation are either query-related or micro-cloud related:
   - A number of query trees
   - Target evaluation criteria
   - Query processing metadata (data volume to be processed, estimated execution time, …)
   - Details on the micro-clouds (cache location, inter-cloud bandwidth/latency)

5. The query trees are stored as JSON files, including metadata for each stage of the query. Metadata are represented as the two parameters k and q and are the basis for our cost calculations (on duration):

   The queries require time to process and generate output:
   k (time to process): h/GB
   q (bandwidth on transport): GB/h

   As an example, to calculate the cost in hours for a query stage including transfer, where *input-size *k* is the processed data that will have to be transferred with a bandwidth *q* if necessary:

$$costs = (inputsize * k) + (inputsize * k)/q$$

   Example queries : We took queries similar as the ones provided by the AMPLab benchmark (AMPLap, 2015) as a basis and adapted them to real use cases of LEADS. These queries are either simple or complex and each operator represents a stage:

   1) simple

```
SELECT sentiment, url FROM webpages WHERE sentiment > 0,
ORDERBY sentiment
```

   2) complex

```
SELECT domainname, AVG(pagerank) FROM webpages, JOIN entities
ON url=webpageurl WHERE entities.name="EC", GROUP BY domain-
name, ORDERBY AVG(pagerank)
```

6. At each stage the data will be always read locally, but the output can be moved somewhere else. Hence, the scheduling decisions can target load balancing of nodes, or shortest execution.

7. The query entry point can be any micro-cloud, the output is moved to the entry point.

3.3 Use Case

Under the assumptions we stated above, the problem of scheduling could be specified by the following simple example.

Let's assume a query having 2 read operations and a JOIN on their results:

- `read left on keyspace "webpage"`
- `read right on keyspace "entities"`

having stage 2 a join (creates a map/reduce job)

Whereas the reads are map-only operations (translated to the MapReduce jargon), the join will also have a reduce phase. The read will be where the data is located (data local), and will load data into the cache. The join will produce a multitude of data, hence it is critical to put the output where it is needed next, already before executing the join. This is the most critical part for scheduling. If there are no follow-up joins, all following operators will work on the produced data by the current operation.

In a simple scenario, there are two possible cases.

1) Both keyspaces of the read operations are on micro-cloud A: The output of the join will be created on micro-cloud A to avoid data movement. All successive operations will be performed on the same cloud, under the assumption that not further joins need to be executed.

2) The keyspace "webpage" is on micro-cloud B, while the keyspace "entities" is on micro-cloud C. Each of these reads will create an intermediate micro-cloud.

   Scheduling decision: Where is the best place to put the intermediate caches as an input for the join?

   a) on micro-cloud B

   b) on micro-cloud C

   c) another micro-cloud

3.4 Simulator

As scheduling query trees depending on huge datasets is a challenging task, we chose to work with two schedulers:

1) simple random scheduling for load balancing, and

2) a smarter scheduler based on neural networks trained with a genetic algorithm that is able to consider also the costs of transporting data. This scheduler is especially interesting as it can also be used in the case of more complex scenarios than started in LEADS (see Section 3.6).

An artificial neural network (ANN) represents a generalized mathematical model of the human brain for computationally demanding tasks. An ANN comprises a network of simplified neurons, where each of them is capable of receiving, processing and transmitting information. The connections between the neurons keep weights, allowing the neural network to adapt according to the input.

A typical design of an ANN is a three-layered-network consisting of input neurons, hidden neurons and output neurons with an information transport from input to output (feed-forward networks).
To get the desired output from the set of input, the weights of the connections between neurons has to be set properly. These weights can be either set manually with the help of domain-specific knowledge, or automatically with the help of learning functions (Abraham, 2005).

One possibility for a learning function is to use genetic algorithms, such as described by Holland (Holland, 1975). A genetic algorithm works on a set of fixed population of candidates and applies the principles of evolution to improve them from generation to generation, until a given goal is reached. In technical terms the genetic algorithm evaluates candidates against a given fitness function. From generation to generation the candidates are sorted according to their fitness and the best x candidates are chosen. These candidates propagate to the next generation. To reach the same population size as the last generation, the remaining candidates are built by mutation, crossover and new randomly generated candidates. For mutation and crossover, random elite candidates are chosen (for more details please refer to (Fehervari, 2013)).

We rely on the FREVO simulation framework (FREVO, 2015), a tool that allows for testing several neural networks and genetic algorithms for networked behavior. The reason for choosing this tool is mainly simplicity. With FREVO, it is only necessary to implement the input and output neurons of a neural network and the fitness function of the genetic algorithm for a simulation.

Error! Reference source not found. shows an overview of FREVO. The tool supports graphical as well as textual output including visualization of the optimization as well as the evaluation stage. In the optimization stage, the tool uses the genetic algorithm to optimize the fitness defined in the implementation. The neural network will be trained during this stage. Later, the best candidate neural network will be used for the evaluation stage, where parameters can be adapted (e.g., number of micro-clouds, number of requests per client, etc.) and the generic fit be tested. The design of the simulation with FREVO is therefore straightforward and can be easily extended for further schedulers or setups needed for LEADS.

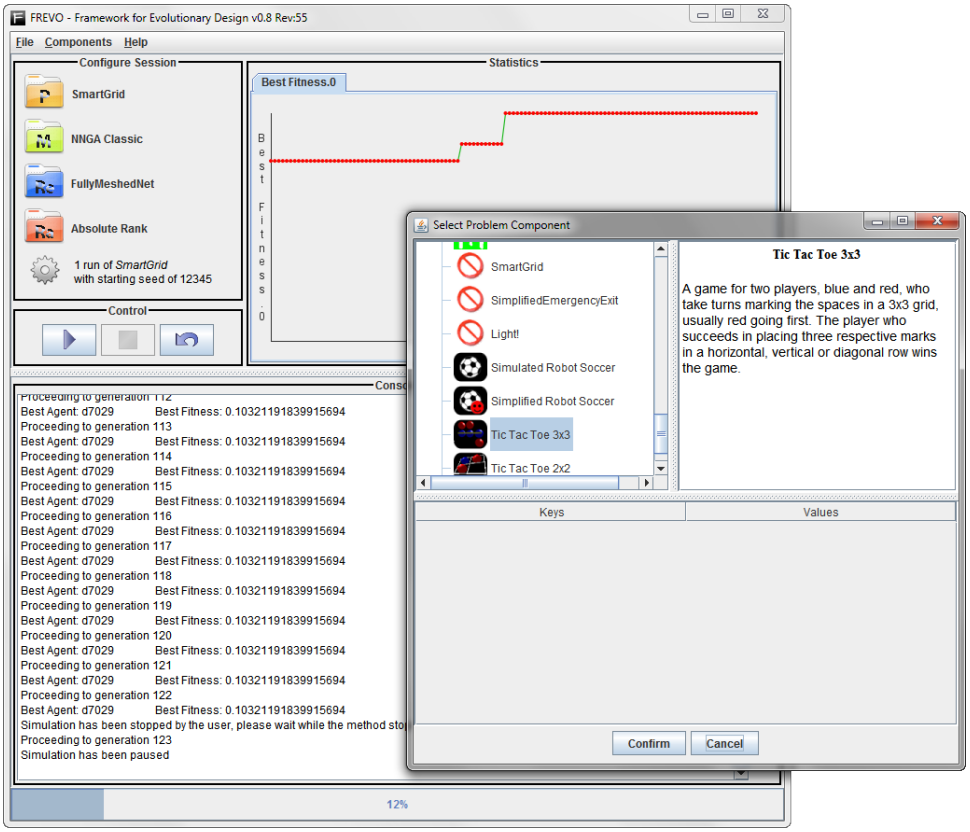

Figure 10 FREVO Tool Overview

3.5 Implementation

For the scheduling the neural network will be consulted at each stage of a query, to decide where the output of a query stage should go.

The output will be scheduled to a micro-cloud, the internal scheduling to specific nodes is covered by *Infinispan*. The current implementation does not limit the number of query stages a micro-cloud can operate on**.** The micro-clouds are fully connected via a fully-switched 10 Mbit/s which resembles the network measurements done on real micro-clouds. Each micro-cloud has a number of clients that send one query after another, picking at random between simple and complex queries.

The ***input neurons*** for the neural network need to be represented as floating point parameters (which is a requirement of FREVO). Our goal is to provide both optimal costs (duration in hours) as well as load balancing for network stability. Therefore, we chose the following input neurons:

1. $(current\ costs_{cloud})/(stages\ served_{cloud})$

The first input neuron covers statistics about the hours (costs) the current cloud has worked on in relation to the already served stages by the same cloud. This input informs about the current efficiency of this micro-cloud.

2. $\text{current tasks}_{\text{cloud}}$

The current tasks per cloud represent the number of currently served query stages (=tasks) by a given micro-cloud. This parameter has an impact on the load balancing among the micro-clouds.

3. $costs_{stage}$

The costs per stage provides relevant measures about the efficiency of a given query. The scheduling process tries to minimize the average costs per query.

Note that the input neurons can be changed, depending on what the goal of the neural network should be. In our case we wanted to focus on the costs in combination with load balancing.

The ***output neuron*** is the micro-cloud identifier that indicates where the next stage should be scheduled to.

The ***fitness function*** should reflect the goals of the system; hence we chose one that only focusses on load balancing and one that also considers the costs (duration in hours):
(load balancing of stages executed per cloud):

$$1/(stdev_{stages})$$

(load balancing of stages and low costs):

$$1/(stdev_{stages} * avg(costs_{stage}))$$

To investigate different system setups, we provide the following ***simulation parameters***:

1. Number of micro-clouds
2. Number of clients per cloud
3. Number of requests per client
4. Storage ratio of keyspaces
5. The input size of a query

3.5.1 Learning Phase

In the learning phase the genetic algorithm performs the simulation under the given setup several thousands of time. Recall that in each generation a set of candidates is evaluated. In our case the simulation is executed with different setups for the neural network at least 50 times per generation. The learning phase was set to 200 generations (which can be extended if needed). Only the best setup for the neural network is represented in the figures below, where we show the Fitness development among 200 generations for two fitness functions. We can see in both cases that the fitness increases quickly, and within a few generations the fitness reaches the maximum value. If we would optimize for more generations on the one hand the system might overcome some local minimum or on the other hand might over-optimize for the given neural network. In the case of the second fitness function, we can see that the it stabilizes for a while. This means we have either reached a local or a global maximum.

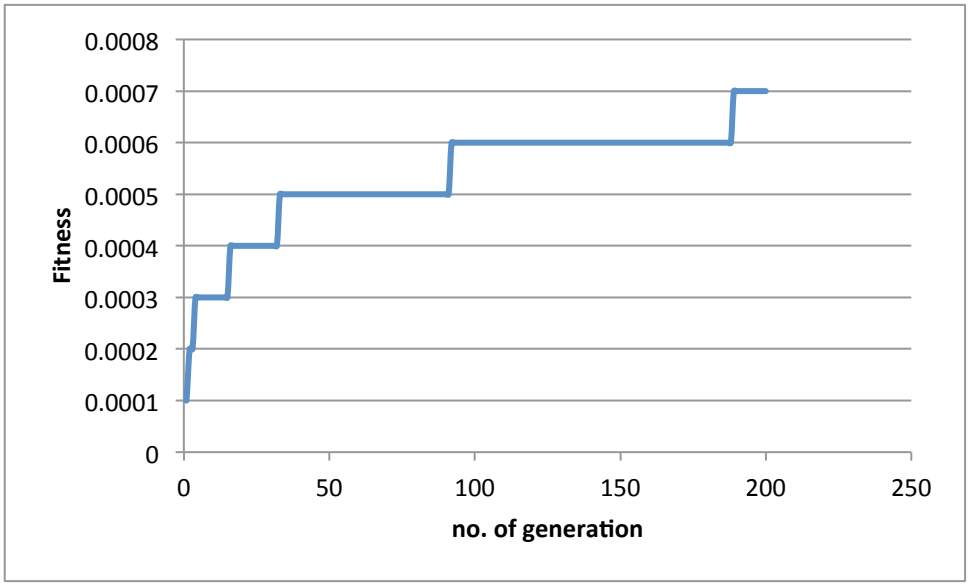

Figure 11 Load balancing

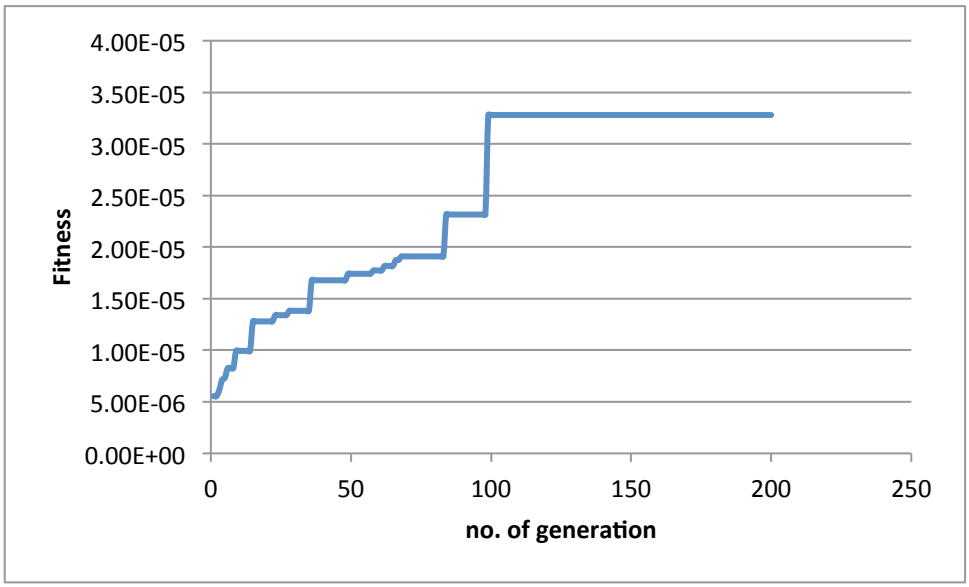

Figure 12 Load balancing and costs

The parameters we used for the learning phase are covering a small scenario, mostly because of performance (the execution time of the simulation if it runs with the genetic algorithm). The size of the storage is configured according to the big data benchmark AMPLab (AMPLap, 2015). In the evaluation, we will show that the similar results can be achieved even when deployed in a larger setup.

inputSizeGB	29
numberOfClientsperCloud	10
numberOfClouds	10
numberofQueriesperClient	100
webpagesRatio	60

3.5.2 Evaluation Phase

For the evaluation, we use the candidate neural network from the last generation evolved. Since we focus on an efficient solution that support both cost optimization and load balancing, we only show the results for the cases optimized with the second fitness function.
In the evaluation phase, we are able to adapt the parameters to see their effect on the neural networks' decisions. In addition, we compare the results against a random scheduler implemented in the same FREVO simulation framework.

First, we compare the costs per hours in relation to the size of the input. Notice that we use 29 GB as default input size for each keyspace. If the k parameter is 1.0 h/GB, the costs for processing 29 GB of input in a read will be at least 29 hours. If transport is added and q is 1 GB/h, additional costs of 29 hours will have to be accounted. Hence, the impact of the location of a join is significant. At the end of our simulation, i.e., when all client requests are processed, we collect the average costs per stage and normalize them by dividing them by the input size. This allows us to compare also the impact of larger input sizes on the costs. Figure 13 depicts 4 scenarios for our schedulers. In the default case we use the same parameters as in the optimization phase. In all scenarios we run the simulation 5 times with different input seeds. Figure 13 presents the weighted average of the results. In general, we can see that the smart scheduler is stable against an increased number of clients and an increased number of queries. Increasing the number of clouds decreases the operating costs, because the number of clouds with a matching keyspace increases. The random scheduler requires higher operating costs per stage due to increased transport requirements for load balancing. An increased input size, however, shows some impact on the costs of the smart scheduler, though it is still comparable with the results of the random scheduler.

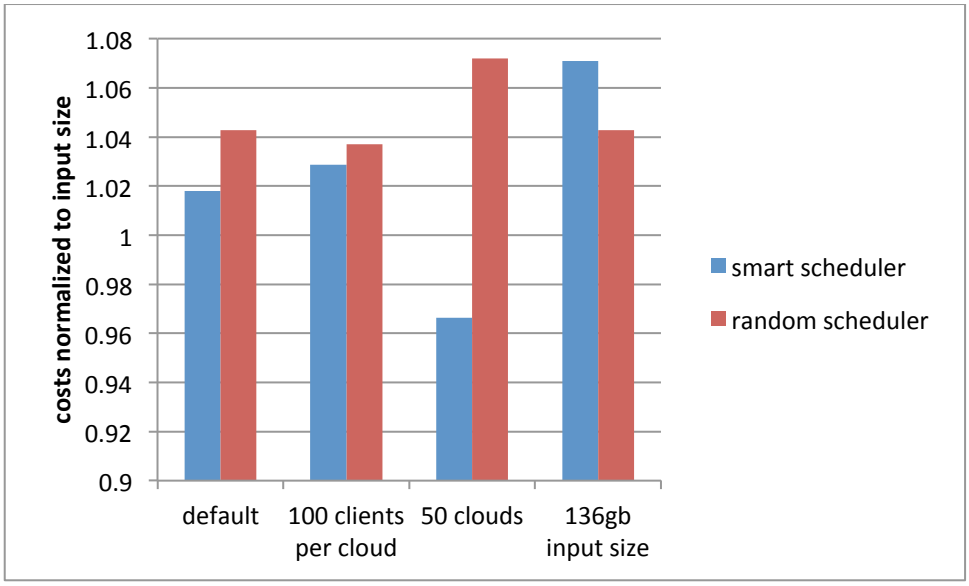

Figure 13 Cost normalized to input size. The parameters of the default case are the once used during optimization.

Regarding the load balancing, the random scheduler represents the optimum with, in average, 4000 stages served per cloud in the default scenario. The averages are the same for the smart scheduler in all cases. It is however more interesting to look at the median to see the goodness of the result.

Figure 14 shows the median of the stages served per cloud for all scenarios. Note that in the case of 100 clients the number of stages to be served would be 10 times higher (i.e., 40,000 instead of 4,000), although for presentation reasons results are divided by 10. In the figure we see that the default case is almost perfectly balancing the stages amongst nodes, while still having lower costs than the random scheduler. Also an increased number of clients do not have a major impact on the results. However, if we increase the number of clouds the neural networks favor cost optimization over load balancing. The change of the input size shows a higher balancing of the costs and the load balancing, which opens opportunities for additional evaluation of different fitness functions.

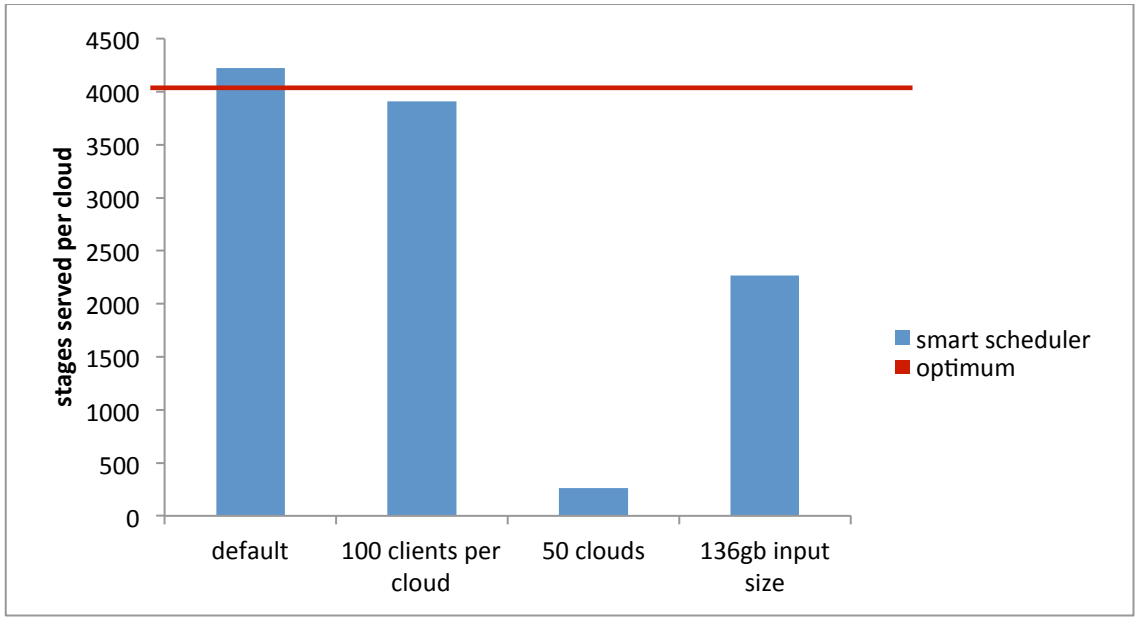

Figure 14 Stages served per cloud (load balancing)

3.6 Discussion and Outlook

The simulation allows for deciding which scheduler policies are necessary in order to operate efficiently on the LEADS stack. The smart scheduler represents a first step towards cost-efficient decisions and opens up for many research opportunities, if some further points are taken into consideration.

1) Heterogeneity of micro-clouds regarding power consumption, costs of operation, hardware capabilities, etc.
2) Covering Ensemble caches (the keyspaces are not replicated as in the current implementation, but are distributed over multiple micro-clouds).
3) Further optimization cases beyond costs and load balancing.

4. Conclusion

In this deliverable, we described the final LEADS data placement and scheduling component developed within the context of WP4 of the LEADS project. We first presented our approach on partitioning the whole web into chunks using a sample crawl. The chunks, i.e., buckets are then assigned to micro-clouds using several heuristics such as proximity as well as storage capacity. In our evaluation, we were able to show that our approach is well suitable for a large number of globally distributed micro-clouds as it results in good data locality.

With regards to the scheduling part, we improved our initial task scheduler by utilizing artificial neural networks in conjunction with genetic algorithms for training. We were able to show that the training is effective as well as the scheduling can save a considerable amount of resources and decrease costs.

5. Bibliography

Abraham, A. (2005). Artificial neural networks. In *Handbook of measuring system design.* Wiley.

Amazon Elastic Compute Cloud. (n.d.). From http://aws.amazon.com/ec2

Amazon Simple Storage Service (Amazon S3). (2015, September 7). From https://aws.amazon.com/s3/

AMPLap. (2015, September 7). From https://amplab.cs.berkeley.edu/benchmark

Common Crawl. (2015, September 7). From https://commoncrawl.org/

Fehervari, I. (2013). *On Evolving Self-organizing Technical Systems.* Dissertation, Universität Klagenfurt, Austria.

FREVO. (2015, September 7). From http://frevo.sourceforge.net/

GeoIP2 City Database . (2015, September 7). From https://www.maxmind.com/en/geoip2-city

Ghemawat, J. D. (2004). MapReduce: Simplified Data Processing on Large Clusters . *OSDI'04: Sixth Symposium on Operating System Design and Implementation.* San Franciso.

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems: An introductory analysis with applications to biology, control, and artificial intelligence. *University Michigan Press* .

WARC file format. (2015, September 7). From http://blog.commoncrawl.org/2014/04/navigating-the-warc-file-format/