SEVENTH FRAMEWORK PROGRAMME

Networked Media

Specific Targeted Research Project

SMART

(FP7-287583)

Search engine for MultimediA environment generated contenT

D5.1.b "SmartReduce" Engine (Public)

Due date of deliverable: 01-11-2012

Actual submission date: 04-12-2012

Revised submission date: 16-01-2014

Start date of project: 01-11-2011 Duration: 36 months

Summary of the document

Code:	**D5.1.b SmartReduce Engine (Public)**
Last modification: **State:**	15/1/2014 Final
Participant Partner(s): **Author(s):** **Fragment:**	GLA & AIT M-Dyaa Albakour, Craig Macdonald, Iadh Ounis, and Aristodemos Pnevmatikakis No
Audience:	☒ public ☐ restricted ☐ internal
Abstract:	*This document is D5.1.1* *Cut-down public version of the D5.1 report*
Keywords:	SmartReduce, real-time indexing, distributed indexing, Indexing efficiency
References:	

1 Executive Summary

1.1 Scope

The main objective of the SMART project is to build an open source multimedia search engine, which provides scalable search capabilities over environment generated content i.e. content captured in the physical world via sensing devices. The SMART framework has been architected as a software stack of three layers in which the search layer is positioned at the centre. The search layer sits at the heart of the SMART framework where it indexes the data originating from the lower layer (the edge nodes) and provides search capabilities to the higher layer (the SMART applications and services).

The SMART search layer therefore plays a significant role within the SMART framework and delivering the search layer is critical to the success of the SMART framework. The SmartReduce[1] engine is our realisation of the search layer's infrastructure where we address the main challenges imposed by the nature of the SMART framework. In particular, by extending Terrier[2] with the required modifications, the SmartReduce engine operates in real-time where it indexes data streams originating from edge nodes as soon as they arrive to provide the freshest results possible as a response to a user query. SmartReduce also builds upon a distributed infrastructure that makes it scalable to handle a larger amount of data streams.

This document is a public cut-down version of the SmartReduce report that presents the architecture of the SmartReduce engine and justifies the design decisions made to provide an efficient and scalable retrieval infrastructure required in SMART. It also summarises the benchmarking results of the efficiency and scalability of SmartReduce. The full version of the SmartReduce report may be obtained by contacting the authors at the University of Glasgow.

1.2 Audience

The SmartReduce engine and the content of this deliverable can be of interest to a wide range of individuals within the following groups:

- **SMART software developers**: The developers of the SMART project, notably those dealing with the open source software implementation of SMART components in the search layer and SMART applications/visualisation libraries.
- **SMART project members:** The deliverable gives insight to all project members involved in delivering the SMART concept. Notably members involved in the development of WP4 and WP6 can better understand how SmartReduce operates and scales to large streams of data. Also they use this deliverable as guidance to use and interact with the Search API.
- **The Open source community:** The open source community, notably the community that will be built around the SMART results, will use the present deliverable as a guide towards understanding the scalable architecture of the search layer, as well as its efficiency performance.
- **The Information Retrieval community**: The challenges addressed in this deliverable are relevant to IR researchers and practitioners. Our solutions advance the state of the art in efficient real-time indexing and retrieval of social and sensor data streams.

1.3 Summary

This deliverable identifies the specific SMART requirements for the SmartReduce engine, to set the scene for the major challenges imposed on its architecture, which can be summarised as providing *real-time* and scalable indexing and retrieval capabilities for large *steams* of sensor metadata. An overview of the search layer is also given to illustrate how SmartReduce supports this architecture.

To support real-time indexing and retrieval in SmartReduce, we have developed a real-time extension

[1] The name SmartReduce is inspired from the MapReduce programming model for the *distributed* processing of large data sets.

[2] http://terrier.org

of Terrier. In this deliverable, we present our extension where we illustrate the various techniques to tackle the challenges of building a robust and an efficient real-time index. Our extension makes it possible to concurrently retrieve the freshest results to a user query while continuously updating the index with latest data streamed from social or sensor networks.

To tackle the challenges of the efficiency and scalability imposed by the large number of metadata streams that need to be indexed and retrieved in SMART, we have built SmartReduce upon an established open source distributed processing framework.. In this deliverable, we identify and review the available stream distributed processing frameworks that can be used for building SmartReduce. The architecture of SmartReduce, built upon open source Storm[3], is then presented highlighting how we addressed the requirements of building an efficient scalable architecture for the real-time indexing and retrieval of sensor and social streams. To this end, the deliverable reports on the performance of SmartReduce in providing an efficient and scalable architecture suitable for SMART.

The deliverable also uncovers the context in which the SmartReduce engine operates within the SMART framework. In particular, typical examples of the metadata streams about the physical world generated by the SMART edge nodes are described.

1.4 Structure

The deliverable is structured as follows:

- We first analyse the requirements of the search layer in Section 2.
- In Section 3, we present how Terrier is extended to operate in real-time, which is a major requirement in SMART.
- Section 4 surveys the available open source real-time distributed data processing frameworks to justify the design decisions made for building the SmartReduce engine. We also summarise the evaluation results of SmartReduce efficiency and scalability in that section.
- Section 5 summarises our conclusions.

[3] http://storm-project.net/

2 Requirements for Indexing & Retrieval in SMART

In this section, we identify the specific requirements for the indexing and retrieval processes in SMART. They stem from the overall functional and non-functional requirements of SMART studied in the requirement analysis phase that is documented in the D2.1 deliverable [Smart-D2.1]. The development of the SmartReduce engine, which is the distributed infrastructure of the search layer, is driven by these specific requirements and some other general SMART requirements. For each requirement, we describe the implications it imposes on the development of the search layer and the SmartReduce engine and the decisions made when implementing the engine.

SMART has identified specific functional requirements for the *search engine* related operations. The following ordered list illustrates these requirements and discusses their implications.

1. *The SMART search engine should produce rankings of events, possibly in response to a user's query (R1.6.1 from SMART deliverable D2.1).* The search layer should implement efficient ranking models for retrieving events relevant to the user query from an index of the metadata produced by the edge nodes. The SmartReduce engine should be able to deploy these ranking models in a distributed manner.
2. *The search engine gathers new input from sensors via edge nodes, and must be robust to communication failures with edge nodes (R1.6.2).* The SMART architecture should define a unified API between the search layer and the edge nodes. SmartReduce should use this API to communicate to the edge nodes and it should tolerate data communication errors so that it does not corrupt the index.
3. *Input from sensors via the edge nodes should be used to model the importance of these events. The current data from a sensor may be compared to the previous background history of that sensor (R1.6.3).* SmartReduce should maintain low-level data from sensors via the edge nodes. They should be represented in the index so that the background information about the sensors can be easily obtained when ranking events for a search query.
4. *As recent events are likely to be more important, the search engine should regularly index new data arriving from edge nodes (R1.6.4).* The search layer should support a real-time operation mode. The SmartReduce engine should be able to index in *real-time* updates from the various edge nodes. As soon as an update arrives from an edge node it will be indexed and made available for search. Also SmartReduce should be able to answer a user query by retrieving events, concurrently while indexing, in *real-time*.
5. *These event rankings should be accessible through an API available to use case applications (R1.6.5).* The search layer should define a RESTful API which allows applications to use the full power of the search capabilities.

There are also Integration-level requirements that have impacted design decisions when developing the SmartReduce engine. In particular:

1. *Reusing existing open source software tools (R1.1.1).* The SmartReduce engine is built upon the state-of-the-art open source Terrier search engine [Ounis2006, Macdonald2012]. Terrier is established as an Information retrieval platform and is widely used in a variety of retrieval applications
2. *Using proven and trusted languages (R1.1.4).* SmartReduce is built with Terrier which is written in Java and other open source tools that are compatible with the Java programming language which is a widely used and an established object-oriented language with plenty of deployment platforms.

In the following section, we describe the architecture of the SmartReduce engine, which addresses the above requirements. The SmartReduce engine provides an infrastructure upon which the various search components are implemented to meet the requirements of indexing and retrieval in SMART as described above. It extends the open source Terrier platform and build upon open source frameworks to achieve this. Most notably, SmartReduce should ensure that the indexing and retrieval in SMART should be real-time where as soon as an update is received it becomes available for search. Section 3 describes the real-time indexing implemented in SmartReduce by extending Terrier traditional on-disk indices. Moreover, SmartReduce offers a scalable environment where the index is *distributed across* multiple index shards (machines) so as to cope with potentially a high number and volume of sensor and social streams. Section 4 describes the distributed indexing in SmartReduce.

3 Real-time Indexing

SMART operates in a real-time setting, where edge nodes continuously process sensor raw data as they are obtained from the physical world, and make the resulting metadata available for search as soon as they are produced. Therefore, it is a requirement that the search layer should be able to provide a response to user queries whilst continuously updating the underlying index with new information to provide the freshest possible results. Providing support for real-time indexing and retrieval in SMART framework is challenging and requires specialised infrastructure for both indexing and retrieval. This is because traditional on-disk indices are designed for static collections where the documents do not change regularly and the indexing process takes place offline in one batch.

In this section, we explain how we extend Terrier with a real-time indexing infrastructure that is capable of performing incremental indexing in which the index is maintained and updated in the timely manner that SMART users would expect. In particular, to enable Terrier to support incremental indexing, fast and efficient in-memory data structures are used for indexing and retrieval. When the in-memory structures fill up available system memory, they are flushed to disk automatically by our implementation. Moreover, the in-memory postings are further compressed to enhance efficiency and allow more documents to be indexed into memory before available memory is exhausted.

3.1 Building In-memory Incremental Indices

An incremental index for dynamic collections, such as the SMART metadata streams, permits new documents to be appended to an online in-memory index without re-building the index or re-indexing the collection. Postings may be retrieved for the processing of a query as soon as the document has been indexed. We build an incremental index that consists of a single in-memory index partition, which is periodically flushed to disk using on-disk index partition(s) as required. The incremental online in-memory index partition and the static read-only on-disk index partitions are wrapped up in a “MultiIndex” structure to make them appear as one index for retrieval purposes.

The in-memory index has methods for indexing a new document, which updates the in-memory index structures with the postings information and a method to force a *flush* the contents of the memory structures to disk. An incremental index has a notion of a *flush policy*, which controls when the contents of the in-memory index partition should be automatically flushed to disk. We provide three flush policies, namely (1) no-flush, (2) flushing after indexing a user-configurable number of documents, and (3) flushing on reaching some user configurable memory usage limit.

Moreover, an incremental index that has been flushed multiple times has several sets of disk-based data structures, and will be less efficient for retrieval than an index of the same document with a single set of disk data structures. However, these sets of data structures can be *merged* to increase efficiency. Moreover, a *merge policy* can be specified, which controls what happens to flushed memory partitions. We develop three different policies for index merging which are (1) “no merge” where simply the flushed memory partitions accumulate on the disk over time, (2) “single merge” where the memory partition is merged with all existing disk partitions to make a single large partition, and (3) geometric merging (as described in [Strohman2006]), which merges flushed indices into an optimal number, order and size.

The in-memory index consists of an in-memory lexicon, an in-memory inverted index, an in-memory metadata index, and other index structures that contain information about the individual documents and the collection as a whole. These in-memory index data structures mirror the components of the Terrier disk-based index structures. In particular, we have implemented two forms, uncompressed and compressed.

Uncompressed In-Memory Index: Throughout the implementation of the in-memory index data structures there are regular uses made of non-standard Java types that have been imported from the open-source Trove[4] library for high performance primitive collections. In particular, this library has been proven to be faster than the standard Java collections (e.g. ArrayList) for primitive types. For instance, the storing of integer numbers within a list, as is required for generating the posting lists in the indexing process. In fact, using standard Java collections would result in an undesirable performance penalty

[4] http://trove4j.sourceforge.net/

caused by the “auto-boxing” of the Integer types in a data structure.

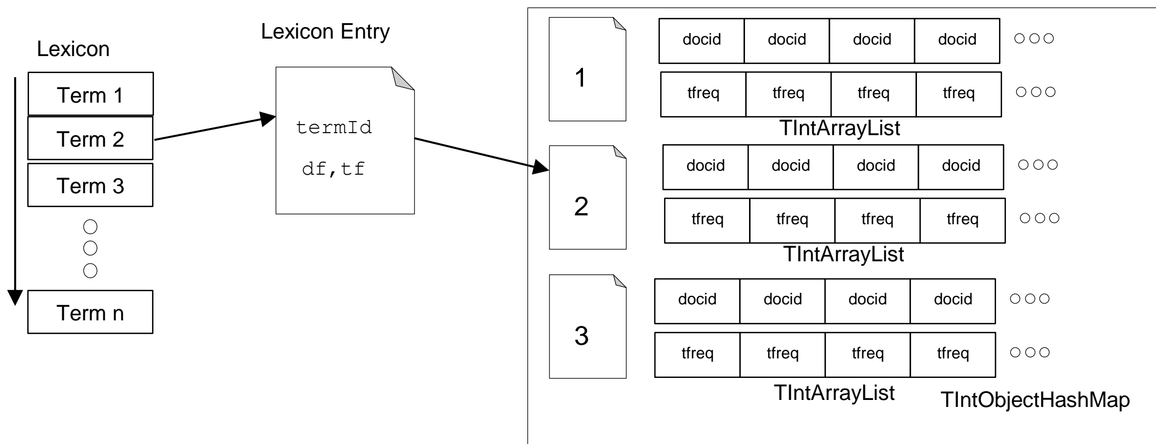

Figure 1: The implemented in-memory data structures

In Figure 1, we illustrate some components of the in-memory index, which are implemented using the Trove data structures (high performance hash maps and array lists). The lexicon is implemented as a (key, value) mapping from a “String” term to a lexicon entry structure. The lexicon entry contains an identifier of the term, its document frequency (number of documents the term appears in) and the term frequency (number of times the term appears in the collection as a whole). The inverted index is implemented using a mapping from term id to posting list, with the posting list being represented as two parallel arrays.

We have also implemented more complex posting types such as those that record the position of the terms in the document (blocks index), and those that record the fields of the document (e.g. title, URL, etc.) in which the term occurs (field index).

Compressed In-Memory Index: To enhance efficiency, the postings in our in-memory index can be further compressed using standard inverted index compression techniques. This increases the amount of documents that can be indexed before flushing the in-memory partition to disk. Moreover, the reduced IO, even from memory, can enhance efficiency by allowing a higher overall throughput of postings during retrieval [Büttcher2007].

As our posting lists in the inverted index are fundamentally composed of lists of integers, the compression is down to minimising the storage required for these lists. Following the standard Terrier compression scheme [Ounis2006], two forms of compression are used:

1. **Elias Unary Encoding** [Elias1975]: Unary encoding represents a positive integer n with n bits. For example, to represent the numbers 1,2,3,4 and 5, it requires five Java integers being stored as 20 bytes, or 160 bits while with Unary encoding it requires only (1+2+3+4+5=)15 bits. We use Unary encoding to encode the term frequencies as the distribution of terms in a collection usually follows a Zipfian distribution where only a small number of terms occur very frequently and a majority of terms are in the long tail, i.e. the majority of terms have low frequencies that require less storage with Unary encoding.
2. **Elias Gamma Encoding** [Elias1975]: Gamma encoding represents a number n with $2.\log_2(n)+1$ bits. Gamma encoding is also optimal for low integers, but differs from unary in that it provides denser encoding when the upper bound is not known. It is used for encoding the document identifiers (docids) as they have a much higher range than term frequencies, potentially much higher as it is dependent on the collection size.

The implementation of compressed inverted indices for memory partitions is similar as for the uncompressed case, however, each posting list of the inverted index is implemented as a list of bytes, which is extended as required for each term. This contrasts with the uncompressed inverted index, which is implemented as two integer lists, one of docids and one for frequencies.

4 Distributed Real-time Architecture

In this section, we describe the novel distributed infrastructure we develop in SmartReduce to perform real-time indexing of data streams in a parallel and distributed manner. MapReduce establishes itself as distributed paradigm to process large amount of data effectively and efficiently. In fact, MapReduce can provide a scalable infrastructure to perform distributed indexing and retrieval of large amount of *static* collections of documents in a batch processing mode [McCreadie2012]. However, MapReduce is not suitable for continuous streams of data such as live posts in social media or stock exchange data. In other words, it is targeted on batch processing applications. Therefore, other parallel data processing frameworks have emerged to handle this type of applications. We first provide a survey of the available parallel streams frameworks. Then we discuss in details the implementation of our distributed infrastructure in SmartReduce highlighting its main components and how they interact with each other. Finally, we perform a thorough analysis on the efficiency of the distributed infrastructure in SmartReduce by conducting a number of experiments to uncover its indexing and retrieval performance under realistic operational settings.

4.1 Survey of Distributed Real-time Processing Frameworks

We focus our discussion on two popular frameworks for parallel processing of continuous streams of data. These frameworks are the open source Apache S4[5] and the open source Storm.[6] Both frameworks provide an environment for stream processing in a distributed fashion. We provide a detailed comparison between them from various aspects to highlight their differences and justify our choice of the processing framework.

S4 started as a research project at Yahoo! Labs in August 2008 out of the need to personalize search ads in real-time. It was then open sourced and eventually moved to Apache Incubator. Storm on the other hand was developed by BackType, which was then acquired by Twitter. Storm was eventually open sourced by Twitter to the community.

Other frameworks exist for high-level stream processes such as Esper[7], Drools Fusion[8] and Flumebase.[9] However, these frameworks are limited in that they are not general-purpose in nature, i.e. they are targeted to specific applications like databases and they are suitable for building custom stream processing infrastructure like the one we are aiming to develop.

4.1.1 A Comparison between Apache S4 and Storm

Table 1 summarises the main advantages and disadvantages of the S4 and Storm frameworks, and assist us in determining the appropriate choice of framework.

In particular, we choose Storm for the development of the SMART search engine (SmartReduce) for the following reasons:

- Intuitive architecture and easier to develop and maintain.
- Transparent task distribution.
- Easier to test and debug.
- Growing user community.
- Guaranteed processing.

---

[5] http://incubator.apache.org/s4/

[6] http://storm-project.net/

[7] http://esper.codehaus.org/

[8] https://www.jboss.org/drools/drools-fusion.html

[9] http://flumebase.org

Table 1: Summary of advantages and disadvantages (denoted + and -, respectively) of the S4 and Storm distributed processing frameworks.

S4	Storm
+ Conceptually powerful	+ Guaranteed processing
+ Automatic load balancing	+ Transparent task distribution
+ In-build client interface	+ Java/Eclipse Debugging
- Complex configuration	+ Active Community
- Opaque processing	+/- Much left 'to the developer'
- Potentially lossy data	- Automatic load balancing is not mature.
- Slow to debug	- Still under development

4.2 SmartReduce Distributed Architecture with Storm

We have developed the SmartReduce engine that forms the backbone of the search layer and allows performing real-time indexing and retrieval of the various streams upon the Storm distributed processing framework. As described above, Storm provides a modular software architecture that allows developers to define the processing components applied on the data streams and assemble them to achieve the required results. With Storm, streams are produced by defining computational units called *spouts,* while the processes on these streams are defined by computational units called *bolts.* A Storm *topology* can be built as a network of spouts and bolts, with each edge in the network representing a bolt subscribing to the output stream of some other spout or bolt. A topology is an arbitrarily complex multi-stage stream computation and topologies run indefinitely when deployed on a Strom cluster. Storm automatically handles the exchange of data between the processing units of the topology (the spouts and bolts).

The SmartReduce engine is built as a storm topology that is depicted in Figure 2. SmartReduce interfaces with the physical world using the SMART Edge Nodes [SMART-D3.1]. SMART Edge Nodes represent the computing layer in SMART that is responsible of processing sensor streams and managing the process of streaming sensor and social metadata to the SMART Search Layer. On the Other hand SmartReduce communicates with applications to submit queries and retrieve results with a RESTful API. The underlying technology used to maintain the distributed index and retrieve results is the in-memory Terrier extension described in Section 3. To guide the reader on understanding how the architecture works, we detail below the three distinct data paths within SmartReduce

1. **Indexing path**: Whenever a new update from an edge node or a post on social media arrives via a data spout, it is passed through to the Accumulator bolt. In this case, the accumulator bolt first updates the global statistics of the index (e.g. total number of documents, global statistics for each term, and so on) and then arbitrarily selects an Index Shard bolt to send the data to. Then, the Index Shard bolt updates the data structures of its in-memory index.
2. **Retrieval path**: Whenever a new query is submitted to the system via an application it is passed through to the Accumulator. The Accumulator sends the query to all the shards along with any required global statistics for answering the query. Each Index Shard Bolt scores the documents it maintains and sends the results to the merger bolt. The merger bolt then consolidates all the answers and sends them over to the application.
3. **Filtering path**: The filtering path is the one that new messages follow from the accumulator to the Running Query bolt. The Running Query bolt decides whether the message is relevant to any registered query. If relevant, the message is pushed to the corresponding application/user.

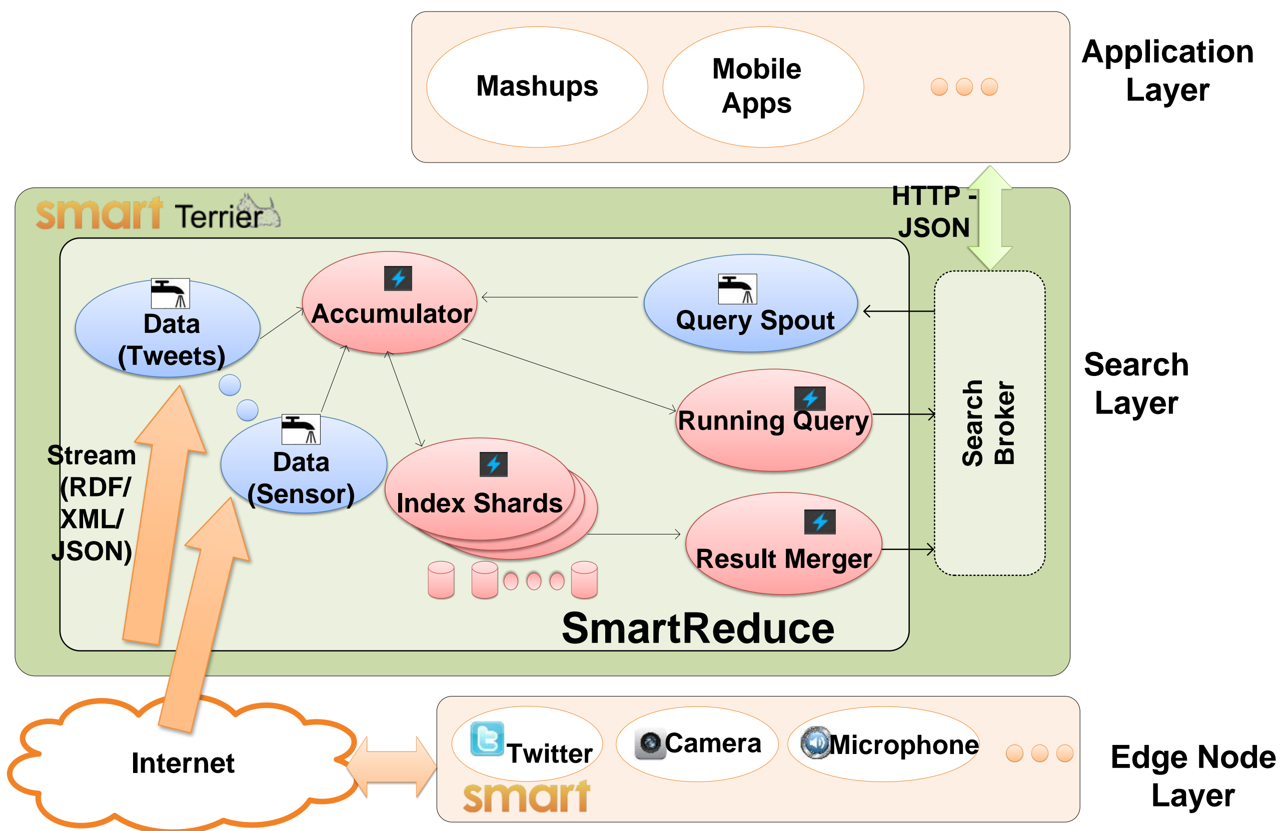

Figure 2: SmartReduce's Storm toplogy (Blue circles represent spouts, while red circles represent bolts)

4.3 Efficiency of SmartReduce

We conducted a thorough benchmarking to measure the efficiency of the distributed framework and its scalability. We conducted a number of experiments to thoroughly evaluate the various aspects of the framework. In particular, the experiments aimed to address the following research questions:

1. How fast is the memory indexing within our distributed framework?
2. How does the distributed architecture scale horizontally with an increased number of shards?
3. What is the average query latency?

4.3.1 Datasets

To achieve an experimental setup with large scale metadata streams, we simulate metadata similar the typical metadata produced by a SMART Edge Node[SMART-D3.1]. As the AIT's edge node typically sends around 2-3 updates per second via the CouchDB HTTP API, we simulate an edge node by producing a random single update every 300ms (3.33 updates per second) similar to the metadata described in [SMART-D3.1]. We simulate 2400 edge nodes by replicating this process and sending 8 random updates every 1ms. In particular, we develop a spout that generates a random update every 1ms and replicates that 8 times in the topology. We also use the social media data from the popular Twitter social network. Although the full stream is not publicly available, the Gardenhose stream – which is 10% of the original stream – is public and is used in our experiments.

To measure the query latency, a realistic stream of queries is required to be fed into the system over time to simulate a real situation in which SMART operates. Since the SMART concept is still at a development phase with a number of different use cases in mind e.g. smart cities and surveillance, we are not able to collect a representative sample of a query load submitted to SMART over time. In other words, we do not know what a typical query log of the SMART system looks like. For this reason and to

simulate a realistic query load, we use query logs obtained from a commercial search engine. In particular, we obtain the query load from the Microsoft RFP (request for proposal) 2006 dataset[10] which was first released by [Craswell2009] to the research community. The dataset contains queries submitted to the MSN search engine for a period of one month in 2006. Each entry in the log records the query submitted, and the timestamp of the query, in addition to other information such as the user session in which it was submitted. The logs reveal that there are different behaviours of search activity throughout a day, where the traffic of queries increases and peaks at certain times (high traffic periods) e.g. midday and decrease at other times e.g. during the night. For our experiment, we select 3,500 queries in a window of 5 minutes from a high traffic period (11 queries/second on average).

4.3.2 Summary of Benchmarking Results

We summarise the results as follows:

- Using simulated sensor observation data, crowd analysis data, we benchmarked the indexing efficiency of SmartReduce with one Indexing shard. We find out that the indexing latency (time for exchanging messages within the distributed framework and index a sensor update) is very low (10 ms) when the number of received messages does not exceed the maximum throughput. With a reference to the requirements of SMART [SMART-D2.1], SmartReduce is on track to meet the requirements for real-time indexing of edge node updates, as long as we do not exceed a certain load of connected edge nodes.
- Using simulated sensor observation data as before, and by varying the number of indexing shards, we benchmarked the scalability of our framework by estimating the maximum throughput (the maximum number of messages per second that can be made available in the index) in each case. When more shards are introduced, we observed the maximum throughput scales in a sub-linear manner. This slightly sub-linear scaling is expected because of the overheads in memory and network communication between the shards. However, we observed that the architecture has a very close performance to perfect linear scaling. This is a promising result and it means that for a real deployment of SMART, only more hardware is required with a cost close to linear.
- Using the Twitter Gardenhose stream (10% of the entire Tweet stream) and a simulated query load from a public web queries dataset, we benchmarked the query latency within the framework. The results show that the system copes well with this relatively high query load, with a mean response time of 171ms. These results are satisfactory since it means that SmartReduce will meet the query latency requirements [SMART-D2.1]. Moreover, we note that the reported response time includes network latencies, which ensures the realism of this benchmarking test. Finally, while the mean response times are very low, we note that as we increase the complexity of the ranking models developed for SMART, we may expect retrieval time to increase slightly.

[10] http://www.microsoft.com/en-us/news/features/2006/may06/05-31livelabs.aspx

5 Conclusions

In this deliverable, we have presented a description of SmartReduce, our realisation of the indexing and retrieval infrastructure in the SMART search layer. SmartReduce addresses the major technical challenges for indexing and retrieval in SMART. In particular, it offers a real-time indexing infrastructure capable of answering user queries with the freshest results while continuously indexing data streams from SMART edge nodes. This is achieved by extending open source Terrier with new in-memory indexing structures. Moreover, SmartReduce relies on a novel distributed framework which makes it scalable to large amount of data streams. This is built upon open source Storm, which acknowledges the wider open source strategy of the SMART framework. Storm is chosen over other emerging frameworks of parallel stream processing for its transparency, reliability and the growing open source community behind it.

This deliverable has also benchmarked the efficiency of the distributed framework and its scalability. The benchmarking results estimate the maximum load that can be handled sufficiently by the framework. They also illustrate the ability of the framework to scale to larger streams (increasing the maximum load) of data by adding more computing resources. In fact, the results assert that the framework can handle large scale loads by adding more hardware with a cost close to linear. In addition, we have estimated the average query response time of the SmartReduce engine by simulating a realistic query load. As summary, our benchmarking procedures show that SmartReduce is on track to fulfil the real-time constraints that are required in a live deployment of the SMART stack.

6 References

[Büttcher2007] S. Büttcher and C. L. A. Clarke. Index compression is good, especially for random access. In Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM 2007). Lisbon, Portugal, November 2007.

[Craswell2009] N. Craswell, Rosie Jones, G. Dupret, E. Viegas. Proceedings of the 2009 workshop on Web Search Click Data (WSCD), Barcelona, Spain 2009.

[Elias1975] P. Elias. Universal codeword sets and representations of the integer. IEEE Transactions on Information Theory, 21(2): 194-203, 1975.

[Macdonald2012] C Macdonald, R. McCreadie, R.L.T Santos, I. Ounis. From Puppy to Maturity: Experiences in Developing Terrier. In Proceedings of the 3rd SIGIR 2012 workshop on Open Source Information Retrieval OSIR, 2012.

[McCreadie2012] Richard M. C. McCreadie, C. Macdonald, I. Ounis. MapReduce indexing strategies: Studying scalability and efficiency. Information Processing & Management 48(5): 873-888, 2012.

[Ounis2006] I. Ounis and G. Amati and V. Plachouras and B. He and C. Macdonald and C. Lioma. Terrier: A high performance and scalable information retrieval platform. In Proceedings of the 2nd SIGIR 2006 workshop on Open Source Information Retrieval OSIR, 2006.

[Qiang2008] Wu Qiang, Burges Christopher J.C., Svore Krysta, Gao Jianfeng. Ranking, boosting, and model adaptation. Tech. Rep. MSR-TR-2008-109, Microsoft, 2008.

[SMART-D2.1] SMART FP7 consortium. Deliverable D2.1, “Detailed Report on Stakeholders Requirements”, 2012.

[SMART-D3.1] SMART FP7 consortium. Deliverable D3.1, “Sensors and Multimedia Data Knowledge Representation”, 2012.