Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D4.2.1
Initial Modeling of the ATB Case Study

Date of document: T10

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **ATB**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Initial Modeling of the ATB Case Study

This document summarises deliverable D4.2.1 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC. within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

Deliverable D4.2.1 presents the ATB Case Study: This deliverable reports on the initial modeling of the structural and functional aspects of the ATB case study and its informal requirements, and details how the case study plans to cover its part of the objectives of ENVISAGE.

List of Authors

Amund Tveit (ATB)
Rudolf Schlatte (UIO)
Thomas Brox Røst (ATB)

Contents

1 Introduction **5**
- 1.1 Offline Search on Mobile Devices . . . 5
- 1.2 On-Device Search Opportunity: Towards Mobile Supercomputers . . . 6
  - 1.2.1 Search: From Desktop Web Service to Mobile App . . . 6
  - 1.2.2 CPU Capabilities on Mobile Devices . . . 7
  - 1.2.3 Storage Capabilities on Mobile Devices . . . 8
  - 1.2.4 What can be stored and indexed with 1 terabyte on-device? . . . 9

2 Architecture Supporting Large-Scale On-Device Search **10**
- 2.1 Large-Scale Offline Search on a Mobile Device . . . 10
  - 2.1.1 Prefix Index based Search . . . 12
  - 2.1.2 Inverted Index Based Search . . . 12
- 2.2 The Cloud Backend Architecture . . . 13

3 Crawling of Data to the Cloud Backend Service **15**
- 3.1 Relevant Resources . . . 16
  - 3.1.1 Content Source Resources . . . 16
  - 3.1.2 Cloud Backend Resources . . . 17
- 3.2 Initial Functional Model . . . 17

4 Processing and Indexing Data in the Cloud Service **18**
- 4.1 Relevant resources . . . 18
  - 4.1.1 Hadoop/Elastic MapReduce Resources . . . 19
  - 4.1.2 General MapReduce Performance Characteristics . . . 19
  - 4.1.3 MapReduce for Prefix Indexing . . . 19
  - 4.1.4 Measuring Prefix Indexing running on AWS Elastic MapReduce . . . 20
- 4.2 Initial Functional Model . . . 21
  - 4.2.1 The Concrete Model . . . 21
  - 4.2.2 The Abstract Model . . . 21
  - 4.2.3 Test for Predictive Power . . . 21
- 4.3 Challenges . . . 22
  - 4.3.1 Measurement challenges . . . 22
  - 4.3.2 Modelling challenges . . . 22

5 Distribution of Indexed Data to Mobile Customers **23**
- 5.1 Relevant Resources . . . 23
  - 5.1.1 Mobile User Resources . . . 23
  - 5.1.2 Cloud Backend Resources . . . 24
- 5.2 The Initial Functional Model . . . 24
  - 5.2.1 The Handset Model . . . 24
  - 5.2.2 The Cloud Backend: The Server Model . . . 24

5.3 Challenges 24

6 Conclusions **25**

Bibliography **25**

Glossary **28**

Chapter 1

Introduction

This deliverable presents the initial modeling of the ATB Case Study: the initial modeling of the structural and functional aspects of the ATB case study and its informal requirements, and details how the case study plans to cover its part of the objectives of Envisage. In terms of the three functional models for i) crawling of data (Chapter 3), ii) processing and indexing of data (Chapter 4), and iii) distribution of indexed and compressed data to mobile search app (Chapter 5), most progress has been made for the processing and indexing model. The functional model for processing and indexing is approaching a general model for Hadoop MapReduce parallel processing that can be applicable also outside of the domain of this case study.

This chapter provides background for offline search on mobile devices, Chapter 2 provides architecture and scale for the mobile app and supporting cloud backend, Chapter 3 provides initial modelling of the crawler in the cloud backend, Chapter 4 provides initial modelling of processing and indexing of data crawled, Chapter 5 provides initial modelling of the distribution of processed and indexed data to the mobile apps so they can be searched in, and Chapter 6 suggests some initial conclusions with relation to relevant Envisage objectives.

1.1 Offline Search on Mobile Devices

ATB works on a **mobile search app** which is named Memkite and which features on-device search indices, a corresponding subscription service, and a supporting **cloud service** to build and distribute indices to the mobile app. Since the app has on-device search indices, it can also work offline independent of Internet services as shown in Figure 1.1 for the query "Laks" (Norwegian word for salmon). The high-level system overview is shown in Figure 1.2, where Memkite provides the cloud backend crawling, the processing and indexing, and the distribution of data to the mobile search app. The system depends on external sources (D0–D2) for data, and app stores for distribution of the app to users.

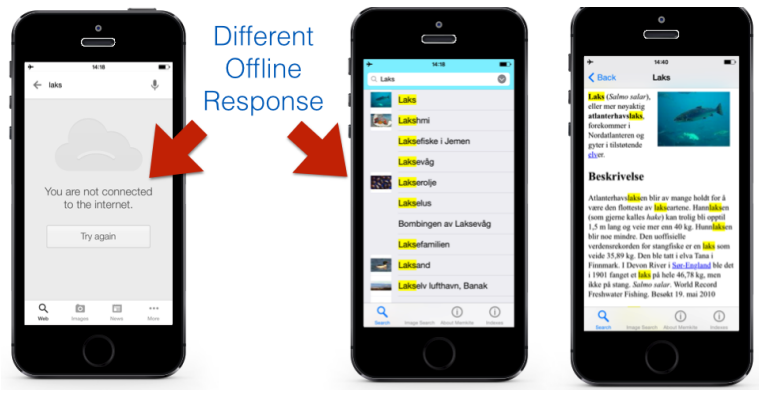

Figure 1.1: Offline Mode: Web Search vs Memkite

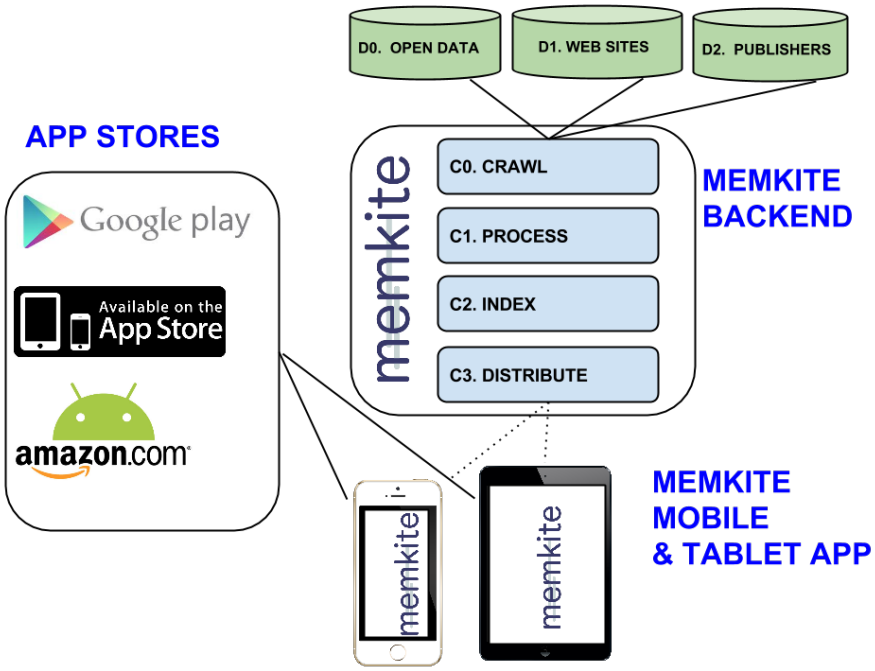

Figure 1.2: High-Level System Overview

The primary motivations for mobile offline search are to:

1. **Increase Search Availability** by not relying on network connectivity for providing search. This means that search can work in situations where mobile network capacity is at strain (e.g., in crowded urban areas), where there is no or poor network capacity (rural areas or in outer space), or in emergency situations such as floods, hurricanes or earthquakes when network infrastructure may not work at all.

2. **Reduce Search Latency** by using consistently fast on-device storage rather than accessing mobile and wifi network with highly variable latency.

3. **Strengthen User Privacy** by being privacy efficient [20] and not transferring or collecting search queries since no query technically needs to leave user's personal mobile device, e.g., strengthen support of United Nations Declaration of Human Rights, Chapter 12.

1.2 On-Device Search Opportunity: Towards Mobile Supercomputers

Mobile phones are approaching supercomputers, and this might change how people will use them for search.

1.2.1 Search: From Desktop Web Service to Mobile App

The number of mobile phone users has rapidly surpassed the number of desktop computer users (Figure 1.3), and an increasing amount of those employ smartphones with computational capabilities rivaling desktop computers. The smartphone capabilities and ease of distribution of software in app stores such as Google's Play for Android and Apple's App Store for iOS, has spawned the development of a large amount of apps.

Time spent on mobile apps is rapidly replacing time spent on the mobile web (Figure 1.4), approaching 9/10 of the time spent in apps (currently 86%). The likely reason for this is that apps provide a better user experience than that of the mobile web wrt. e.g., UI and latency, since they i) utilize the device capabilities with natively compiled code compared to the interpreted code in the browser and ii) better utilize fast local hardware on the mobile device.

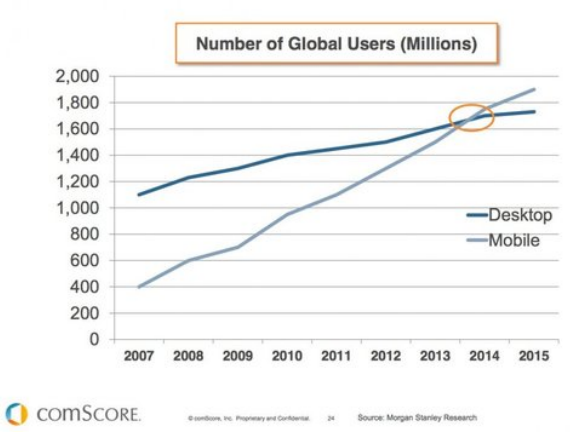

Figure 1.3: Mobile surpasses Desktop

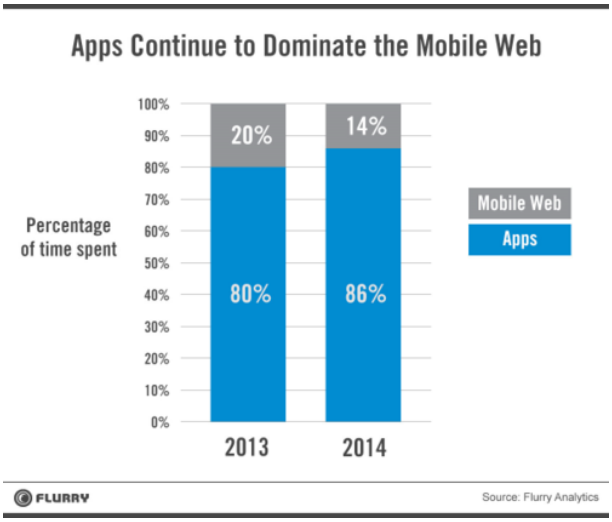

Figure 1.4: Apps replacing Mobile Web

Enabling large-scale on-device *search* for mobile, wearable and tablet devices requires significant on-device CPU and storage resources, e.g., CPU to decode posting lists or decompress documents and low-latency storage with enough capacity to store and retrieve search index data on the device. These CPU and storage resources are becoming available on-device as shown in the next sections.

1.2.2 CPU Capabilities on Mobile Devices

The performance of CPUs and GPUs on smartphones and tablets is accelerating towards matching what could be found in current desktop/server computers and supercomputers from the recent past.

Marc Andreessen, the inventor of the Web Browser (Mosaic and Netscape), points out that his latest smartphone (an iPhone 5S) is faster than a Cray Supercomputer that cost 10 Million dollars 20 years ago (Figure 1.5). Marc's phone has a 64 bit A7 Cyclone CPU combined with a GPU capable of high-performance vector processing with Apple's Metal API [2, 6]. Since Apple's launch of the first 64 bit mobile CPU, there have been several announcements of 64 bit and massively parallel CPUs for mobiles [12]:

1. Qualcomm announced mobile 64 bit CPUs Snapdragon 808 and 810
2. Intel announced the 22 nanometer 64 bit Atom CPU

Figure 1.5: iPhone faster than Cray Supercomputer, Tweet Posting March 16, 2014

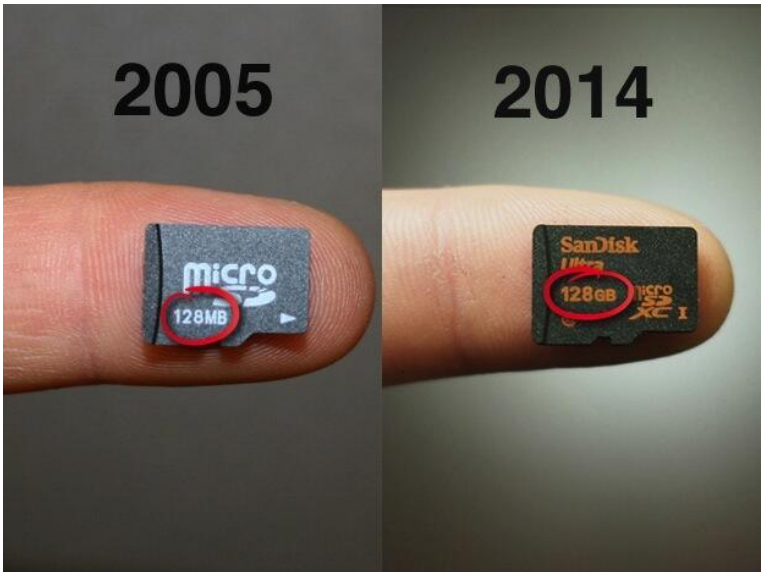

Figure 1.6: 1000-fold increase in storage on MicroSD cards in 9 years

3. Samsung announced the Exynos 5433 64 bit CPU

4. Nvidia announced Tegra K1 CPU with 192 parallel cores

These CPUs, as exemplified by the Apple A7 CPU, are capable of handling of a billion document-sized indices in a fast way. Section 2.1.1 and 2.1.2 discuss the experiences gathered by ATB concerning large-scale prefix index and inverted index on an iPad Mini.

1.2.3 Storage Capabilities on Mobile Devices

Storage capacity on mobile devices is perhaps the fastest progressing hardware technology today. In less than 10 years, from 2005 to 2014, there has been a 1000-fold increase in storage capacity on MicroSD cards from 128 MB to 128 GB (Figure 1.6), and you can now purchase more than 1 terabyte on a USB stick. To put that in perspective: 1 terabyte is more storage than the search engines of Altavista and Google had in their computer clusters in 1999 [3, 21], and the USB stick with SSD storage has approximately 1/100th of the random access latency of the hard drives used in the web search clusters (Figure 1.7).

While Flash/SSD-based storage used in MicroSD cards provides large mobile storage (approximately 700 GB per cubic cm), two technologies can replace it [16]:

1. RRAM: potential of storing 1 terabyte per square cm chip, with up to 20 times faster retrieval rates than SSD [15]

2. Memristor: potential of storing hundreds of terabytes on a mobile device, combined with CPUs developed with Memristors to be able to process all that data rapidly [13]

Figure 1.7: More data on USB stick in 2014 than Web Search Cluster in 1998/99

Further into the future one can potentially utilize DNA based storage, where theoretically multiple petabytes can be stored per gram (700 terabytes per gram has already been demonstrated [1]); to put that into perspective, a current smart phone typically weighs around 100 grams.

1.2.4 What can be stored and indexed with 1 terabyte on-device?

In terms of the amounts of data that it is possible to index and store on a device with 1 terabyte RRAM, it would likely to be technically feasible to have the following on-device [17]:

1. Large knowledge sources such as Wikipedia, Quora and Stackoverflow
2. The last 10 years of **all** published academic papers (text only)
3. Five hundred thousand non-fiction books, which is the rough estimate of all English-language non-fiction book published the large publishers: McGrawHill, Reed Elsevier, Springer-Verlag and Pearson Education
4. Mobile app stores, e.g., Windows 8, iOS App Store and Google Play, which have less than 10 million apps combined
5. 3 million images and thumbnails from Wikipedia

Chapter 2

Architecture Supporting Large-Scale On-Device Search

This chapter describes the architecture of the ATB case study and provides more detail on the on-device search problem the architecture needs to support.

The goal for the use of the Envisage tools in the ATB case study is to support the cloud backend architecture in order to:

1. simulate and analyze (cloud) cost-effects of code changes and changes in cloud configurations
2. simulate and analyze SLA-effects of code changes
3. provide (soft or hard) guarantees on reliability with automated unit test generation.

The use of Envisage tools is planned to be a part of the Continuous Deployment System for the Memkite system as a system test, see part 4 of Figure 2.1. The system needs to pass the Envisage-based tests before the cloud backend is automatically deployed to a staging environment or production.

2.1 Large-Scale Offline Search on a Mobile Device

In order to enable large-scale offline search on a mobile device, the mobile app needs support from the cloud service to do i) crawling, ii) processing and indexing and iii) distribution of indexed and processed data to the mobile app as shown in Figure 1.2. These three cloud services are modelled in Chapters 3–5, respectively.

In order to better understand what the cloud services need to support, the two main types of search and corresponding indices are presented in Sections 2.1.1 and 2.1.2.

The on-device architecture for both search types is shown in Figure 2.2, where

- Layer 1 is the runtime environment, which is a mobile or wearable operating system such as Android, iOS, Tizen, Firefox OS or Windows 8);
- Layer 2 is the app deployment (or distribution) mechanism, which makes use of app stores (e.g., iOS or Google Play);
- Layer 3 is the (potentially encrypted) file system on the device;
- Layer 4 deals with the updates of search indices, either on-device indexing or retrieval of indices and corresponding data from the cloud service; and
- Layer 5 is the main search index library (e.g., prefix and inverted index).

The main search index is written in C++ for portability across platforms, i.e., for integration with Objective-C/Swift on iOS, Java on Android, C# on Windows 8, Javascript on Firefox OS and C++ on Tizen.

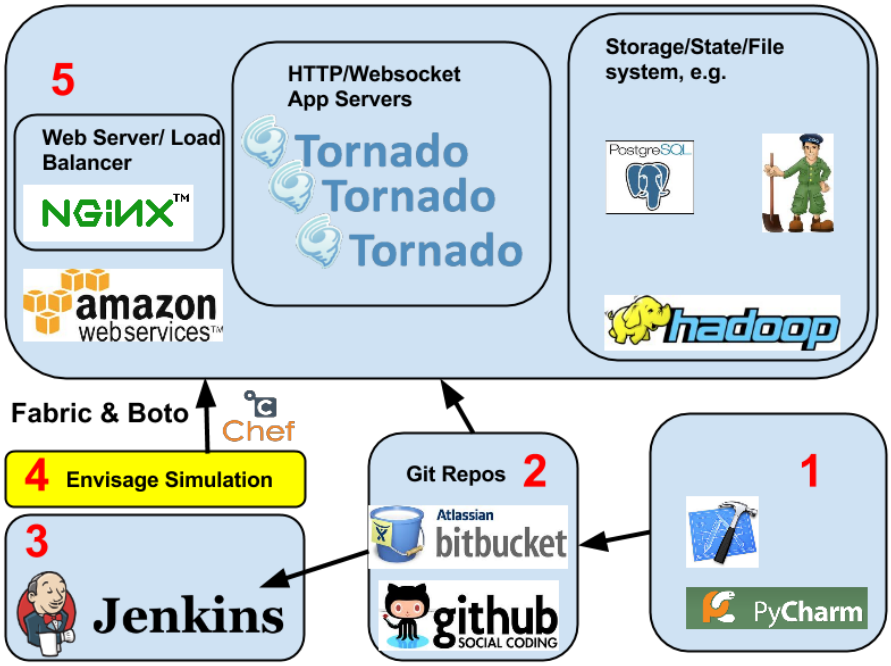

Figure 2.1: Continuous Deployment System

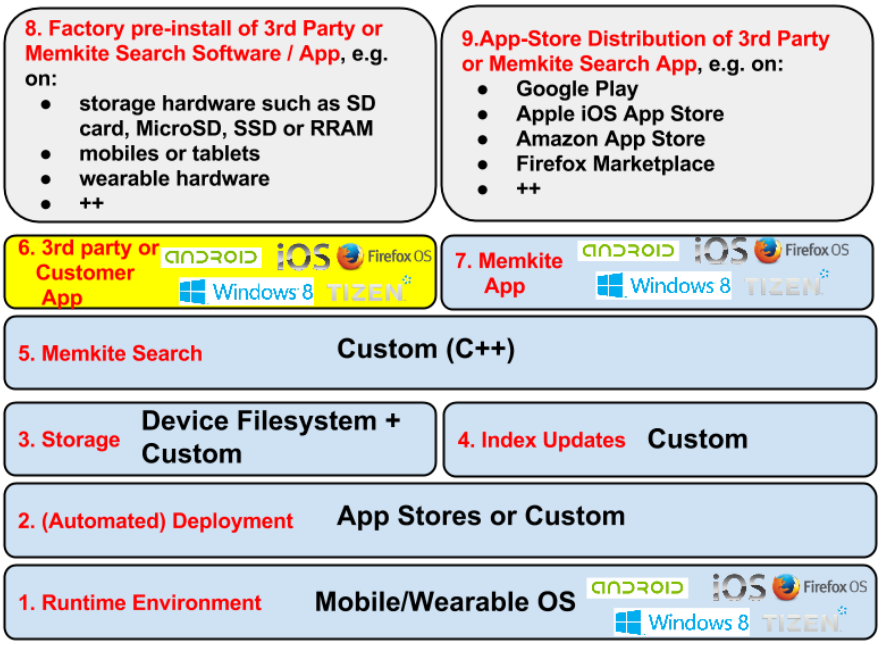

Figure 2.2: Layered On-Device Search Architecture

2.1.1 Prefix Index based Search

Search can be seen as a key-value lookup problem where the key is the textual query consisting of potentially several words or phrases and the search result, which represents the value, is a list of object identifiers (e.g., URLs and titles of documents). Since the upper bound on the number of potential queries is exponential in the size of the alphabet and in query length, storing all key-value combinations in search seems incomprehensible. However, in practice, it might be feasible for the following reasons:

1. **Most search queries are short.** In a web search query log published by AOL, 81.26% of queries have <= 13 characters and 95.95% of queries have <= 23 characters. In a 2009 study of query logs from Google Web Search, queries on iPhone and other mobile phones had an average number of words of 18.25 and 15.89 [8]. Both of the above query logs were done prior to prefix-based instant search, where search results are predicted and appear before the query is fully typed. In a 2012 study of instant search, the average peak query length was found to be 8.89 [4].

2. **Prefix-based indices reduces query lengths.** Consequently, it also reduces the size of the information that the search engine needs to store. For example, if the user types a query such as *amaz* or *univ*, the search engine can predict results such as *amazon, amazing*, ... and *university, universe, universal*, ..., respectively. Since search results typically consist of only 10 individual results, one can look into the length required of a prefix to be the prefix of <= 10 other queries. In the case of the AOL web query log, 99.9% of the prefixes (of queries) of length 18 are in fact prefixes of <= 9 queries. With even longer prefixes, the number of queries of which they are prefixes continues to drop. In fact, 99.9% of prefixes of lengths 19, 20, and 21 are prefixes of 8, 7 and 6 queries, respectively.

3. **Storing a very large set of prefixes is feasible.** To estimate the storage capacity needed to store an index with a large set of prefixes, one may use the n-gram collection from Google based on analysis of millions of digitized books [11] as a guideline. For English n-grams of length 1 to 5, the combined compressed size for the published data set [5] is 803.5 GB (distributed over the length of the n-gram as follows: 4.8+65.5+218.1+293.5+221.5). In terms of a search-related problem (genre classification with machine learning) a recent result reports that using n-grams of length 5 provided the best classification accuracy [9], also better than longer n-grams. This may imply that in practice 5-grams is a good upper bound length for identifying text in search.

4. **Storing a very large set of prefixes with search results is feasible.** ATB experimented with constructing a prefix-index from English Wikipedia with more than 500 million prefixes using MapReduce in the cloud service (see also Chapter 4). This resulted in a 60GB compressed index, which contained both the prefix and the corresponding search result (a list of up 10 Wikipedia titles per prefix). This 60 GB prefix index was extended with approximately 10 GB in compressed document store and 13 GB of compressed images and thumbnails that appeared in the documents, and used for search on an iPad Mini with 128 GB SSD-based storage.

2.1.2 Inverted Index Based Search

Prefix search does not solve all search cases, e.g., it does not support queries with words that do not appear close or next to each other in documents (but still occur in the same documents). The most common approach is to use a posting list based index, also called inverted index. Since inverted indices have sorted lists of URIs per word in the dictionary, it means these indices can be strongly compressed (much better than prefix indices) with techniques such as Variable Byte Encoding, Group Varint, and Elias γ code [14].

Compared to the key-value based lookup, this is much more computationally expensive. For each individual word in the query, one must look up the posting list (compressed set of URIs), decompress the posting list, and merge the posting list for each term. This is in particular tedious for long posting lists. When indexing 1 billion documents one could naively face the problem of decoding and merging posting lists with

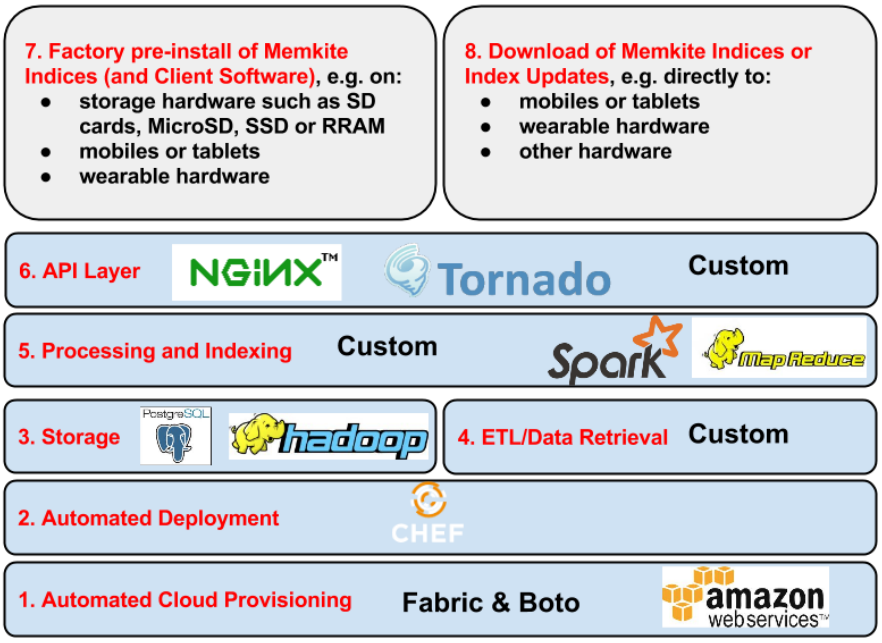

Figure 2.3: Layered Cloud Backend Architecture

ten to hundreds of millions of URIs each. This should happen within a maximum of a few hundred milliseconds, which is particularly hard to do on a resource-constrained mobile device [14]. Common techniques to address this problem, is to use i) segmentation of posting list to speed up merging (not needing to merge all segments) [10], and ii) GPUs to decode and merge posting lists [22].

In the search experiments conducted by ATB, we were able to merge search results from 5 posting lists, representing a query with 5 words, of length 100 million on an iPad Mini with a 64 bit A7 Cyclone CPU in 25 milliseconds.

2.2 The Cloud Backend Architecture

Recall that the entire system has four main parts (Figure 1.2): 1) web and content sites with data; 2) the cloud backend that crawls data from content sites, processes and indexes it and sends it to apps; 3) the mobile app store where users can find and install the app from; and 4) the search app itself.

The cloud backend layered architecture is shown in Figure 2.3.

- Layer 1 is the cloud provisioning which allocates cloud resources such as, e.g., storage and virtual machines. This is separated from deployment since provisioning is tightly coupled to a cloud platform such as Amazon Web Services or Azure, while deployment is more generic once the cloud resources have been allocated.

- Layer 2 is the deployment, which consits of installing, configuring and starting services.

- Layer 3 is the storage. Since we use a mix of small/medium-scale data sources (e.g., Wikipedia and custom data sources for customers) and big data sources (e.g., web crawls), both PostgreSQL and Hadoop are used. Since we are using Hadoop and MapReduce on Amazon Web Services, the HDFS file system is run on top Amazon S3 distributed key-value store.

- Layer 4 is the ETL and Data Retrieval layer responsible for getting data into the storage layer.

- Layer 5 is for processing and indexing. We are currently using Hadoop/MapReduce, but we will move to Hadoop/Spark instead since it is a faster alternative. In the ATB case study, both MapReduce and Spark will be considered.
- Layer 6 is the API layer where various web services related to, e.g., crawling and the distribution of data, run. It is also used to expose APIs in lower layers.
- Layers 7 and 8 are logical layers concerned with distribution. In some cases Memkite data might be preinstalled on mobile or storage devices and in other cases data might be downloaded from the cloud service itself. Consequently, the API layer needs to support both ways of distributing data.

Chapter 3

Crawling of Data to the Cloud Backend Service

Crawling of data from content sources to the cloud backend is necessary for building indices that can be searched on the mobile device. There are three main content sources (see Figure 3.1, or Figure 1.2 for the full system context):

D0. Open Data. Example content includes Semantic Web, DBPedia, Wikipedia, Geographical Data, and Public Statistics. These data are typically accessible over HTTP/HTTPS, FTP/FTPS or RSYNC/SSH protocols, and might require performing queries to get the data (e.g., SPARQL to retrieve RDF/XML). Formats are frequently JSON, XML or HTML. In some cases, CAPTCHA or passwords may need to be handled in order to access content. Depending on the content format, there might be non-standard hyperlinks that need to be detected and crawled. For Semantic Web/RDF data, it may make sense to combine several fine-grained RDF records into one document as part of the crawl process.

D1. Web Sites. Example content includes web pages, text, images, audio, and video. These data are typically accessible over HTTP/HTTPS or SPDY protocol. Some content may only be accessible if the crawler understands and runs Javascript, i.e. behaving as a complete web browser, this may programmatically be solved by using browser-behaving libraries and tools very similar to what is being used in UI testing as part of continuous integration. See ATB's article [19] for an example of programming of browser-behaving component PhantomJS.

D2. Publishers. Example content includes books, newspapers, and academic papers or reports. These data are typically available over HTTP/HTTPS or FTP/FTPS protocol and usually require login and password credentials. Formats may include PDF, HTML, XML or proprietary formats (e.g., from a content management system).

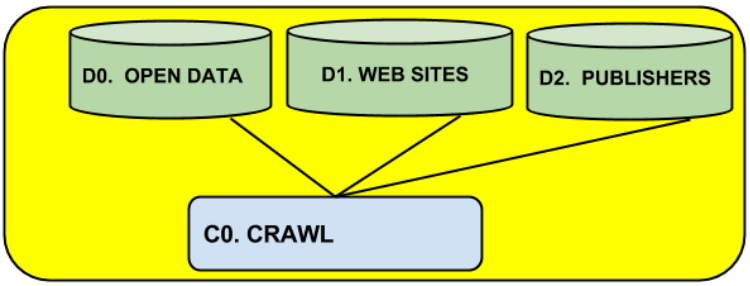

Figure 3.1: Crawling of Data to Cloud Service

3.1 Relevant Resources

Since crawling is about discovering and retrieving data on content sources that are external to the Memkite cloud backend, crawling also spends resources of the content sources. This needs to be considered to provide a complete model of resource use. In the case of network and storage, there is a symmetry in resources used by the content site and the cloud backend: the content site needs the same amount of outbound bandwidth to transfer a content object as the cloud backend needs in inbound bandwidth, and similar for storage of the object.

3.1.1 Content Source Resources

The primary computational resources a Content Source (e.g., a Web Site) consumes in order to serve content to a crawler are as follows:

A. Storage. For storing objects over time, this might be given by the virtual machine in case of a cloud architecture.

B. CPU. For example, virtual machine hours to run web server and storage service (e.g., a database) to make content objects available.

C. Outbound bandwidth. Needed to transfer content objects to crawler.

D. Memory. Memory is used by the web server, this resource might be given by the virtual machine in case of a cloud architecture.

E. Availability distribution. Many web servers have poor uptime availability, and some may be periodically turned off for upgrades or updates.

We do not necessarily know the costs involved for the Content Source. There may be several other components, particularly in case they are hosting the content source themselves (as opposed to serving it in the cloud). These are harder to model and probably out of scope for Envisage.

F. Load Balancing. Hardware or software running on separate machines.

G. Power. The availability of this resource might vary with time-of-day, day-of-year. In Germany there have been examples that on sunny days when solar production is at a peak, you can actually get paid to use power. One can imagine this being used by cloud computing or data center vendors in order to start up and offer very cheap virtual machines when it happens.

H. Security. Both computational security (e.g., Firewall) and physical security of the data center

I. System administration. Human resources needed to keep the system running.

There are also resources/characteristics related to the content itself, e.g.,

J. Content Update Frequency and Distribution. Content may be updated at consistent times, randomly, or never. There may be new URLs or updates to existing ones (e.g, need for recrawl on known URLs)

K. Content Type and Size Distribution

L. Quality, Authenticity of Content. The stealing of content as well as web spam is a common problem. Frequently this is automated and often automatically mixed together. For example, several textual sources may be combined to make the content look authentic and unique (it would fool an NLP parser), although it is basically just random text of no interest.

The above resources assume that Content Sources are independent (except K), but it is very common to serve many content sources from the same machine or cluster with the same IP (for example, web servers on different domain names). This is typically called "Virtual Host" in popular Web Servers like nginx and Apache. This may need to be taken into account for crawl and recrawl scheduling, to avoid providing a huge load on a single IP address that appears to be different web servers.

3.1.2 Cloud Backend Resources

Crawling is bandwidth and storage intensive, and, if postprocessing steps are included as part of the crawl, it also consumes significant CPU.

A. Storage. For storing objects over time, this might be given by the virtual machine in case of cloud.

B. CPU. For example, virtual machine hours to run web server and storage service (e.g., a database) to make content objects available.

C. Inbound bandwidth to transfer content objects to crawler.

D. Memory used by the crawler.

3.2 Initial Functional Model

An initial functional model has been created in the ABS language [7] that models the essentials of the crawling task as described above, with public web sites as the content source. As a first step, we abstract away from the task of parsing the concrete content of web pages. Instead, a web page is modeled via the characteristics essential for crawling (size, outgoing links, last modified date).

```
data DocHead = Head(Int lastUpdated, Int size);
data Document = Document(DocHead head, List<Url> references);
```

Similarly, a web server is modeled via its essential functions (getting the head and body of a URL).

```
interface WebServer {
    Maybe<DocHead> getHead(String path);
    Maybe<Document> getPage(String path);
}
```

Finally, the crawler is initialized with a list of seed URLs and recursively collects document information.

```
interface Crawler {
    Unit addSeeds(List<Url> urls);
}
```

After being augmented with resource costs (bandwidth, CPU, memory) and active behavior in the web server (i.e., updating the last-modified attribute of pages), we plan to use this model to evaluate different crawler strategies wrt. content freshness, cost, resource consumption, etc.

Modelling the characteristics of some other main categories of content sources is also on the plan ahead, such as news sites, blogs, company or organization homepages, and personal homepages.

Chapter 4

Processing and Indexing Data in the Cloud Service

When data has been crawled from content sources, it needs many types of processing before it can be sent to the mobile device. This includes data cleaning, filtering, transformations, entity extraction, natural language processing, clustering, classification, ranking, compression, and indexing (making it searchable). Finally, the index and the corresponding compressed content can be sent to the mobile device. As shown in Figure 4.1, processing and indexing constitutes the intermediate stage in the cloud backend (Figure 1.2 shows the full system context).

In the basic case of HTML or XML content that has a unique URI, the data is transformed into text and then indexed, typically be prefix index or inverted index as presented in Sections 2.1.1 and 2.1.2. This involves indexing large amounts of data, and, in the case of prefix-indexing, a heavy processing of data. Therefore, it makes sense to use reliable parallel computing tools like Hadoop with Spark or MapReduce (see also the cloud backend layered architecture in Figure 2.3).

4.1 Relevant resources

In the ATB use case, we have chosen to use Amazon Hadoop/Elastic MapReduce since it provides an auto-configured Hadoop. However, the performance and cost of jobs can still vary based on the number of resources and configurations.

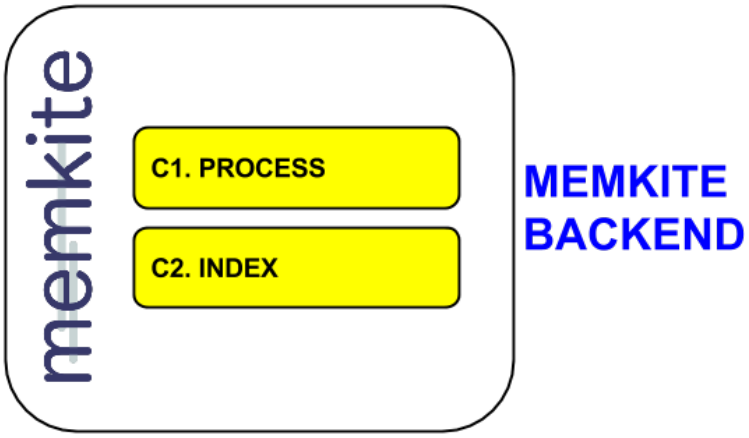

Figure 4.1: Processing and Indexing of Data within a Cloud Service

4.1.1 Hadoop/Elastic MapReduce Resources

A. HDFS distributed file system: the logically distributed file system.

B. Distributed Key Value Store: HDFS runs on top of this; in the AWS case we use Simple Storage Service (S3).

C. Number of map tasks per virtual machine

D. Number of reduce tasks per virtual machine

E. Number of virtual machines used for mappers: the total number of reduce tasks = number of reduce tasks per machine * the number of machines.

F. Number of virtual machines used for reducers: the total number of reduce tasks = number of reduce tasks per machine * the number of machines.

G. Number of input files: the number of files on HDFS (S3) needs to be larger than the number of mapper tasks.

H. Virtual machine type used for mappers specifies CPU, RAM, IO Bandwidth (low, medium or high), and local disk.

I. Virtual machine type used for reducers specifies CPU, RAM, IO Bandwidth (low, medium or high), and local disk.

4.1.2 General MapReduce Performance Characteristics

In a previous study by ATB on the MapReduce parallel implementation of machine learning [18] (*Tikhonov Regularization with a Square loss function*), we experienced the following about MapReduce performance:

1. **Mapper increment sizes can matter a lot:** The amount of work done by the mapper can have a multiplying impact on the work that the reducer needs to do. In practice this means keeping the number of yields in the mapper down, and also balancing the number of mappers with the number of reducers.

2. **Avoid reducer bottlenecks:** Sometimes it can make sense to chain several MapReduce jobs, so one has reducers in the first job that create a partial result before passing it on to the final reducer. A similar approach is to use combiners, which are reducers that run on the same machine as the mapper.

4.1.3 MapReduce for Prefix Indexing

A prefix index is a key-value store from prefix (of a word, phrase or sentence) to a set of URIs that may be ordered by rank. With MapReduce one first finds all prefixes and then outputs the corresponding URI for each (unique) prefix. In addition there might be constraints on the minimum and maximum length of prefixes. There might also be corresponding data one wants to add to the index (e.g., byte positions in files for document and image data, and additional field support, e.g., the document title). This typically adds only a constant factor to each map call and has been left out in the Python pseudocode below.

Mapper pseudo code for Prefix Indexing

```
def mapper(uri, document):
  # 0. Find all positions in a document where a prefix can start
  # and generate all prefixes possible from each position
  for i, character in enumerate(document):
    if isWordSeparator(character):
      for j in range(min_prefix_length, max_prefix_length):
```

```
        if i+j < len(document) and i+1 < len(document):
          prefix = document[i+1:j+1]
          uniqueprefixes[prefix] = uniqueprefixes.get(prefix, 0) + 1

  # 1. Output all unique prefixes
  for prefix in uniqueprefixes:
    yield prefix, uri
```

Reducer pseudo code for Prefix Indexing

```
def reducer(prefix, uri_iterator):
  # 0. get all uris for a prefix
  uri_array = []
  for uri in uri_iterator:
    uri_array.append(uri)

  # 1. optionally rank the uris
  ranked_uris = rank_uris(uri_array)

  # 2. output all uris
  yield prefix, uri_array #ranked_uris
```

4.1.4 Measuring Prefix Indexing running on AWS Elastic MapReduce

In order to be able to model the prefix index MapReduce performance with ABS, there is a need to measure actual performance. AWS Elastic MapReduce provides some measurements in the form of coarse-grained counters at the job level:

1. Bytes read: Files, S3 and HDFS
2. Bytes written: Files, S3 and HDFS
3. Number of tasks: mapper and reducer
4. Memory: virtual and physical memory
5. CPU time spent

However, these metrics are not sufficient to model the prefix index case. In addition, we needed to measure the following metrics:

1. the distribution of call time per map and reduce call,
2. the distribution of input and output bytes per key from the mapper,
3. the distribution of input and output bytes per value from the mapper, and
4. the distribution of number of yield calls per mapper.

There were unfortunately a few challenges measuring the new metrics:

- measurement added an extra toll to the job since a lot of data needed, and in fact made the job fail in the reduce phase due to out-of-memory because of the measurements (Heisenberg-like effect). This was solved by sampling down measurements taken along the way, i.e. storing measurement samples all the way to the reduce step is finished in order to get the distribution measurements above.
- using the MapReduce framework counters is the way to do measurements in MapReduce. This is a limited resource which was not fit for dynamically creating counters depending on data, so the job could actually fail due to running out of number of counters. This was solved by outputting a fixed set of counters that provided a reasonably good statistical description of the distribution per value independent of data size; for example, average, median, 1, 5, 25, 50, 75, 95 and 99 percentile.

4.2 Initial Functional Model

We have modelled a prefix indexing job in ABS, reflecting how the Memkite system runs to prepare crawled data for searching on the client. This models the transformation of a list of pairs (filename and content) into a list of index entries (prefix and filenames).

The model exists in two versions:

- a *concrete model* which reflects the actual computation, and
- an *abstract model* that abstracts from computation but remains faithful to the control and data flow.

Both versions model a MapReduce-based computation and can be used to simulate different deployment strategies, resource models, etc. Both models rely on two interfaces, one for the Worker and one for MapReduce:

Worker Interface

```
interface Worker {
  // invoked by MapReduce component
  List<Pair<OutKeyType, OutValueType>> invokeMap(InKeyType key, InValueType value);

  // invoked by MapReduce component
  List<OutValueType> invokeReduce(OutKeyType key, List<OutValueType> value);
}
```

MapReduce Interface

```
interface MapReduce {
  // invoked by client
  List<Pair<OutKeyType, List<OutValueType>>>
    mapReduce(List<Pair<InKeyType, InValueType>> documents);

  // invoked by workers
  Unit finished(Worker w);
}
```

4.2.1 The Concrete Model

The *concrete model* computes all intermediate and final results and data structures. It can be used for, e.g., static analysis, testing, and resource analysis that rely on having concrete data available.

4.2.2 The Abstract Model

The *abstract model* reflects the control and data flow of a MapReduce-based indexing job but elides the manipulation of concrete data structures. The motivation for the abstract model is to efficiently enable the simulation of *large* input sizes. This will enable direct comparison of simulation results and measurements from concrete MapReduce jobs on the Amazon AWS infrastructure, as run by ATB. The abstract model will typically be run with varying resource models, serving as a test bed for experimenting with resource modeling of CPU, bandwidth, storage, etc.

4.2.3 Test for Predictive Power

Both models can be used to simulate varying deployment scenarios. Deployment scenarios vary both in the number and capacity of machines assigned to the MapReduce job. One expected result of the case study will be an evaluation of the predictive power of the models wrt. the cost of running an indexing job on different numbers of machines on Amazon AWS.

4.3 Challenges

4.3.1 Measurement challenges

In the current version of the models, the distribution metrics are not measured pairwise; we cannot accurately measure pairwise correlations between, e.g., the call time and the number of yield calls in order to get more precise input to build a better model. We further plan to investigate how the instrumentation of MapReduce jobs can be generalized in order to obtain better data for evaluating the predictive power of the modelling approach (Section 4.2) to simulate any type of MapReduce job.

4.3.2 Modelling challenges

We plan to generalize the model to make it easy to simulate and analyze any type of MapReduce job, and not only the prefix indexing of the use case.

Chapter 5

Distribution of Indexed Data to Mobile Customers

The distribution of indexed and compressed data to the mobile app is necessary in order to provide updated data the user can search in offline. This is the final step in the backend, as shown in Figure 5.

5.1 Relevant Resources

In addition to the resources for distribution related to the cloud backend, we need to model the resources of the receiving users with the mobile search app.

5.1.1 Mobile User Resources

A. User Bandwidth and Latency which may depend on their data subscription plan (which could be maxed out on the number of gigabytes to download that month), location (e.g., abroad with constrained mobile network), the settings on their phone (e.g., roaming on or off), the type of network to which they are connected (e.g., EDGE, 4G, LTE or Wifi), the time of day (perhaps following a weekly schedule), or perhaps the context or situation in which they are situated.

B. App Settings and app version: how often they want updates, and which updates they should.

C. App and Index State: what data they have on the phone, and possible index synchronization issues. The fragmentation of the index, which may need merging on the device.

D. Device and Operating System Type: whether the device is a tablet, an Android or iOS phone, and whether it has a GPU.

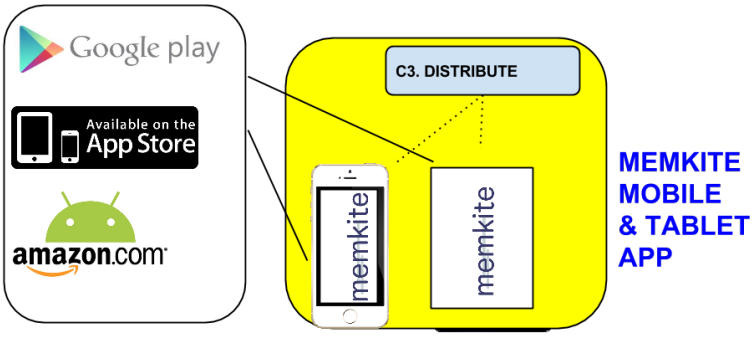

Figure 5.1: Distribution of Indexed Data to Mobile Customers

5.1.2 Cloud Backend Resources

A. Outbound bandwidth to transfer index elements to the app

B. CPU to schedule and perform index update to apps (push), or react when mobile users requests update

C. Memory to keep track of states of mobile users, as input to scheduling and updating

5.2 The Initial Functional Model

An initial functional model of the handset and download server behavior has been created.

5.2.1 The Handset Model

The handsets are modeled as ABS objects that keep a connection of varying bandwidth with the server, notifying the server when bandwidth availability changes. For the first version of the model, bandwidth changes are modeled arbitrarily.

Handset Interface

```
interface Handset {
    Unit receiveChunk(Chunk chunk);
}
```

5.2.2 The Cloud Backend: The Server Model

The cloud backend's data distribution service is modeled as an ABS object which maintains a set of data chunks of varying sizes. These data chunks need to be sent to each handset.

The Server Interface

```
interface Server {
    Unit notifyBandwidth(Handset handset, Int bandwidth);
}
```

5.3 Challenges

With time, CPU, and bandwidth resource models, the functional model of the distribution task will allow the simulation of the effects of running the Memkite system wrt., e.g., cost, battery, and bandwidth for the handset owner, and wrt. , e.g., cost and bandwidth usage on the server side.

Chapter 6

Conclusions

This deliverable has presented our work on initial models for the three parts of the ATB case study:

1. Crawling,
2. Processing and Indexing, and
3. Distribution.

These models are still preliminary as they are preparing the ground for modelling the resource management aspects of the case study. In particular, we have experimented with two versions of the indexing model, in order to accomodate scalability with respect to the size of the data which will be tackled in the resource modeling phase. In the deliverable, we have also discussed how we expect to make the resource management explicit in the models during the next phase of the case study.

Bibliography

[1] Sebastian Anthony. Harvard cracks DNA storage, crams 700 terabytes of data into a single gram. http://www.extremetech.com/extreme/134672-harvard-cracks-dna-storage-crams-700-terabytes-of-data-into-a-single-gram, August, 2012.

[2] Sebastian Anthony. Apple's A7 Cyclone CPU detailed: A desktop class chip that has more in common with haswell than krait. http://www.extremetech.com/computing/179473-apples-a7-cyclone-cpu-detailed-a-desktop-class-chip-that-has-more-in-common-with-has
March, 2014.

[3] Jeff Atwood. Google hardware circa 1999. http://blog.codinghorror.com/google-hardware-circa-1999/, May, 2005.

[4] Inci Cetindil, Jamshid Esmaelnezhad, Chen Li, and David Newman. Analysis of instant search query logs. In Zachary G. Ives and Yannis Velegrakis, editors, *WebDB*, pages 7–12, 2012.

[5] Adam Gray. Google books N-Grams. https://aws.amazon.com/datasets/8172056142375670, January, 2012.

[6] Apple Inc. Metal programming guide. https://developer.apple.com/library/prerelease/ios/documentation/Miscellaneous/Conceptual/MTLProgGuide/MetalProgrammingGuide.pdf, June, 2014.

[7] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer-Verlag, 2011.

[8] Maryam Kamvar, Melanie Kellar, Rajan Patel, and Ya Xu. Computers and iphones and mobile phones, oh my!: A logs-based comparison of search users on different devices. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 801–810, New York, NY, USA, 2009. ACM.

[9] K. Pranitha Kumari, A. Venugopal Reddy, and S. Sameen Fatima. Web page genre classification: Impact of n-gram lengths. *International Journal of Computer Applications*, 88(13):13–17, February 2014. Published by Foundation of Computer Science, New York, USA.

[10] Ronny Lempel and Shlomo Moran. Optimizing result prefetching in web search engines with segmented indices. In *VLDB*, pages 370–381. Morgan Kaufmann, 2002.

[11] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K. Gray, The Google Books Team, Joseph P. Pickett, Dale Holberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin A. Nowak, and Erez Lieberman Aiden. Quantitative analysis of culture using millions of digitized books. *Science*, 2010.

[12] Anthony D. Nagy. Samsung Exynos 5433 tops Snapdragon 801 and 805 in AnTuTu benchmark. http://pocketnow.com/2014/06/23/samsung-exynos-5433, June, 2014.

[13] James Niccolai. Hp says 'The Machine' will supercharge Android phones to 100TB. http://www.techspot.com/news/53497-new-resistive-ram-packs-1-tb-of-storage-into-a-single-chip.html, June, 2014.

[14] Lars Martin S. Pedersen. Postings list compression and decompression on mobile devices. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2013.

[15] Tim Schiesser. New resistive RAM packs 1 TB of storage into a single chip. http://www.techspot.com/news/53497-new-resistive-ram-packs-1-tb-of-storage-into-a-single-chip.html, August, 2013.

[16] Amund Tveit. Mobile eats the cloud. http://memkite.com/blog/2014/04/22/mobile-eats-the-cloud, April, 2014.

[17] Amund Tveit. Technical feasibility of building hitchhiker's guide to the galaxy, i.e. offline web search - part i. http://memkite.com/blog/2014/04/01/technical-feasibility-of-building-hitchikers-guide-to-the-galaxy-i-e-offline-web-search-part
April, 2014.

[18] Amund Tveit. Parallel machine learning for Hadoop/Mapreduce. http://atbrox.com/2010/02/08/parallel-machine-learning-for-hadoopmapreduce-a-python-example/, February, 2010.

[19] Amund Tveit. Continuous deployment for Python/Tornado with Jenkins, Selenium and PhantomJS. http://atbrox.com/2013/05/28/continuous-deployment-for-pythontornado-with-jenkins-selenium-and-phantomjs/, May, 2013.

[20] Amund Tveit. Privacy efficiency - measuring and improving. http://blog.amundtveit.com/2014/05/privacy-efficiency-measuring-and-improving/, May, 2014.

[21] English Wikipedia. Altavista. http://en.wikipedia.org/wiki/AltaVista, July, 2014.

[22] Fan Zhang, Di Wu, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. Fast lists intersection with bloom filter using graphics processing units. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 825–826. ACM, 2011.

Glossary

Terms and Abbreviations

API Application programming interface.

ETL Extract, transform, load: A data warehousing term that describes the process of getting data into a database or a data warehouse.

MapReduce A programming model for processing large data sets in a parallel, distributed manner.

N-gram A string sequence of n characters, collected from a text corpus. N-grams of size 1 are knowns as "unigrams", size 2 as "bigrams" and size 3 as "trigrams".

Posting list In information retrieval, a list of document IDs. A simple inverted index is a dictionary of terms where each term is linked to a posting list.

SPARQL Simple Protocol and RDF Query Language: A query language for data stored in the Resource Description Framework (RDF) format.

SPDY An open networking protocol for transporting web content.

Swift A programming language developed by Apple for iOS and OS X. The language is designed to replace Objective-C.

Tizen A Linux-based operating system for embedded devices.

URI A uniform resource indicator (URI) is a string used to identify a resource. The most common type of URI is the URL, which identifies web resources.