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

Deliverable D4.2.2
Resource Aware Modelling of the ATB Case Study

Date of document: T22

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:
Resource Aware Modelling of the ATB Case Study

This document summarizes deliverable D4.2.2 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 D4.2.2 presents the ATB Case Study. This deliverable reports on the resource aware modeling of the ATB case study.

List of Authors

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

Contents

Chapter 1

Introduction

This deliverable presents a resource-aware model of the ATB Case Study: the ABS model of the structural and functional aspects of the ATB case study and its requirements. The main structural changes in this report compared to the previous report D4.2.1 is more emphasis on mobile device modeling and less on the modeling of crawling, this to better reflect changes in the actual system we try to model. The crawling part in the actual system is simplified; it no longer focuses on broad crawling of an unknown number of web sites and web pages, but rather on periodically checking a given set of sources for new data (see figure 2.2 and 2.1 in chapter 2 to see the architecture changes). Since the resource-aware developed ABS-model is a single end-to-end model (see yellow parts of figure 2.1), it is presented in one chapter instead of being split up over several chapters as in D4.2.1.

1.1 Offline Search on Mobile Devices

This chapter provides background for (large-scale) offline search on mobile devices and expectations to the ABS model. Chapter 2 provides the architecture and scale for the mobile app and supporting cloud backend. Chapter 3 provides the resource-aware ABS model of the entire system, while Chapter 4 offers conclusions in relation to relevant Envisage objectives.

ATB works on a **mobile search app** named Memkite 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 1.2 for the query "Laks" (Norwegian word for salmon). In Figure 1.1 the app has been extended with on-device Deep Learning based image recognition combined with on-device search, in this case searching for a rose with the mobile camera. The high-level data flow from the cloud service (backend) to mobile device overview is shown in Figure 1.3, where Memkite provides the cloud backend with the pro-

Figure 1.1: On-Device Image Recognition driven Search

Figure 1.2: Offline Mode: Web Search vs Memkite

cessing and indexing and the distribution of data to the mobile search app. The system depends on external sources for data, and app stores for distribution of the app to users; see chapter 2 for more information about architecture.

1.1.1 Motivation for Offline Search

The primary goals 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 Wi-Fi network with highly variable latency.
3. **Strengthen User Privacy** by being privacy efficient [11] 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).
4. **Improve Utilization of Mobile Devices** by using fast processing (GPU and CPU) and storage (SSD) available on mobile devices instead of network services with higher latency. Individual mobile devices in 2015 have the capacity of web search clusters in recent past (Figure 1.4).

1.1.2 Search Capabilities of a 1 TeraByte Mobile Device?

As mobile devices are approaching 1 TeraByte of storage - what are they able to store on-device? [12]:

1. Large knowledge sources such as Wikipedia, QA-sites like 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

The hardware - processing and storage - capabilities of mobile devices is further described in appendix A.

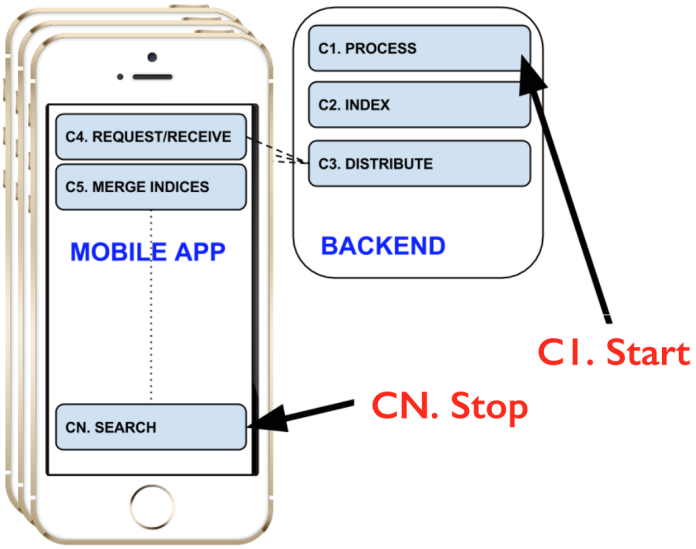

Figure 1.3: System Data Flow

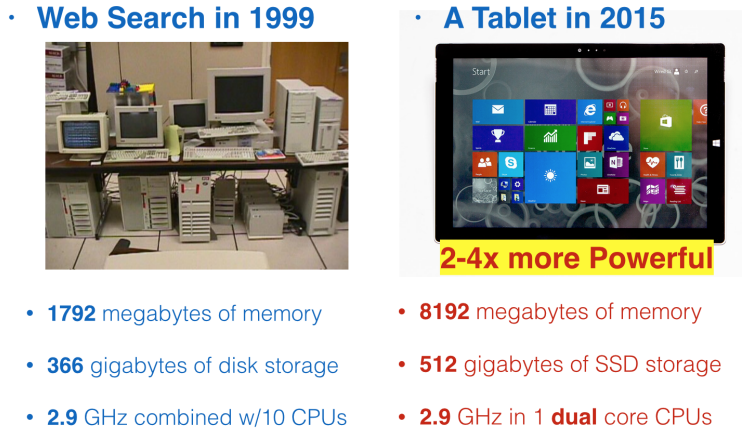

Figure 1.4: Hardware: Web Search 1999 vs Mobile (tablet) 2015

1.2 Expectations and how the ABS Model can help meet them

A key motivation for creating a resource-aware ABS model of the entire search system (cloud backend and mobile search app) is to improve overall system quality and get accurate estimates on resource use (e.g. expensive cloud computing resources) prior to using them. This aligns well with expectations both from customers and ATB (as a service provider).

1.2.1 Customer Expectations

Customers - users of the mobile search app - can expect the following from the app:

Freshness — it will always have as fresh data as possible, i.e. the time from S0 (data arrived in Cloud Backend) to S8 (indexed data is ready to be searched on Mobile Device) in figure 2.1 should be as short as possible. This expectation is matched by an Service Level Agreement (SLA).
ABS Model: this time can be simulated for sets of cloud backend and mobile device configurations

Correctness — it will always work (e.g. independent of on-device index updates).
ABS Model: concurrency errors is one of the most frequent and severe cause of bugs; with ABS-based deadlock analysis combined with code generation this can potentially be reduced

Cost Efficiency — it will use (costly) mobile bandwidth efficiently.
ABS Model: it can simulate with customer bandwidth cost models - e.g. mobile data plans - for individual users and simulate costs over time

Privacy — it will not store or distribute customer search queries.
ABS Model: it can perhaps be proven with Envisage tool Key-ABS that search queries never leave the mobile device

1.2.2 Service Provider Expectations

Cost Efficiency — cloud computing makes it very easy to over-allocate resources but finding a good initial cloud configuration can be hard.
ABS Model: it models cloud resources (virtual machines) with a set of resources per machine - this can be used to find a reasonable configuration through measurements from simulation with various simulated cloud resource configurations

Performance — in a distributed system even a small code change might have a big impact on performance.
ABS Model: In a tuned model doing changes there and simulating the performance effect before doing the actual code changes

Software Quality — Similar as for correctness expectation from customers, but from a software development and maintenance perspective
ABS Model: use Envisage tools for automatic unit test generation, code generation and deadlock analysis to improve software quality

Automation — automated continuous deployment of code to production
ABS Model: integrating the model with a continuous integration server to simulate cloud computing costs prior to doing deployment and stopping deployment if the simulated cost change is too high/expensive

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.

2.1 Changes in Architecture

In the initial cloud architecture the focus was on supporting flexible crawling of content (e.g. arbitrary-sized web crawl), but this is no longer the case since the data in the business use cases is more limited (typically just a single or a couple of data sources). This change is shown in figures 2.2 (before) and 2.1 (now). The yellow parts show what is modelled with ABS.

2.2 Cloud Backend Architecture

Recall that the entire system has four main parts (Figure 2.2): 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 (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 consists 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 of the Amazon S3 distributed key-value store.
- Layer 4 is the ETL and Data Retrieval layer responsible for getting data into the storage layer.

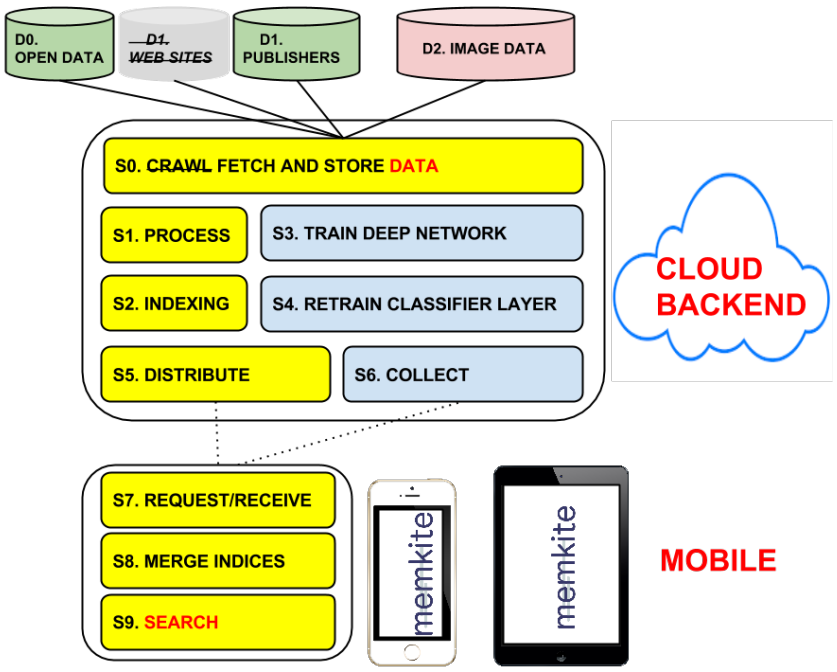

Figure 2.1: Up-to-date System Overview - yellow parts modelled with ABS

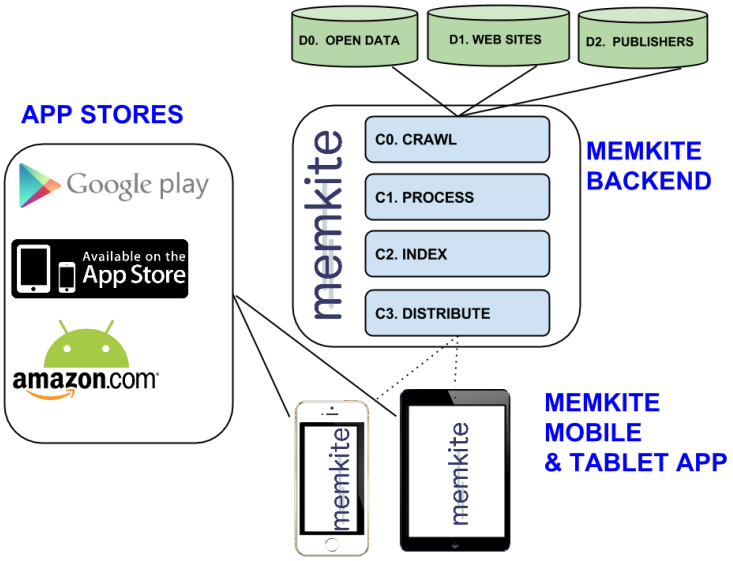

Figure 2.2: Past System Architecture - presented in delivery D4.2.1

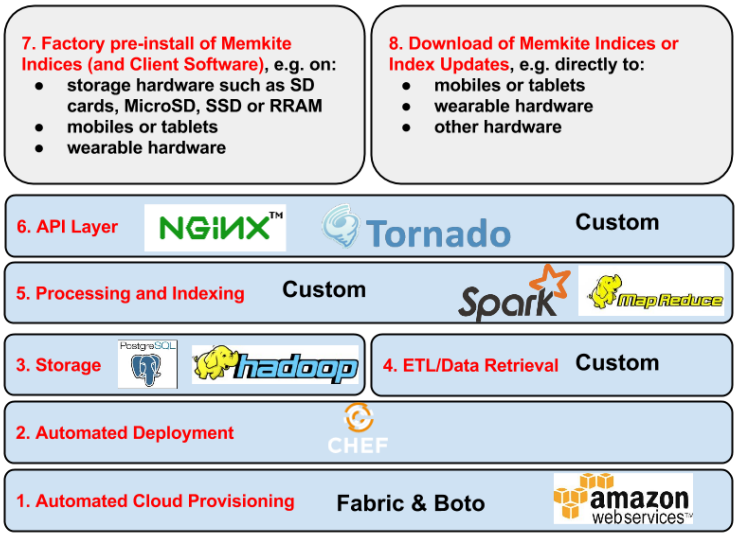

Figure 2.3: Layered Cloud Backend Architecture

- Layer 5 is for processing and indexing. We are currently using Hadoop/MapReduce (*which is modelled with ABS in the ATB case study*), but we consider moving to Hadoop/Spark instead since it is a faster and more cost-efficient alternative.
- Layer 6 is the API layer where various web services related to, e.g., fetching and distributing 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.

2.3 Mobile Device Search Architecture

In order to enable large-scale offline search on a mobile device, the mobile app needs support from the cloud service to do i) data fetching, ii) processing and indexing and iii) distribution of indexed and processed data to the mobile app as shown in Figure 2.1. This is modelled with ABS and described in Chapter 3.

In order to better understand what the cloud services need to support, the main types of search and corresponding indices are presented in Section 2.3.1.

The on-device architecture for both search types is shown in Figure 2.4, 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).

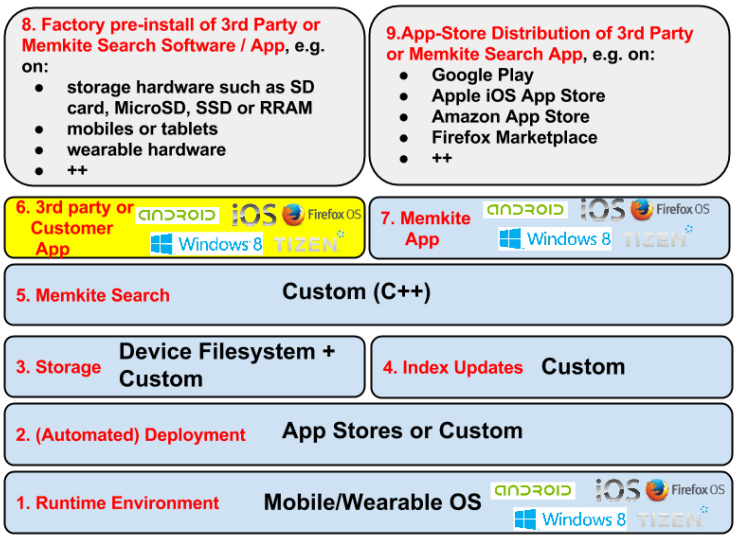

Figure 2.4: Layered On-Device Search Architecture

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.

2.3.1 Inverted Index-Based Search

Memkite search is a combination of prefix-based (direct lookup) and inverted index search, with inverted index being the most flexible since it is independent of which query length to support Since inverted indices have sorted lists of URIs per word in the dictionary, the means these indices can be strongly compressed with techniques such as Variable Byte Encoding, Group Varint, and Elias γ code [8].

2.3.2 Experiment with large-scale on-device search

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, merge the posting list for each term and finally rank the results. 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 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 [8]. Common techniques to address this problem are to use i) segmentation of posting list to speed up merging (not needing to merge all segments) [5], ii) GPUs to decode and merge posting lists [14], or iii) RAM-based index caches (which are hard to do on a resource-limited mobile device).

In the mobile device search experiments conducted by ATB, we were able to merge search results from 5 segmented 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 - **posting lists of length 100 million can correspond to a 1 billion document indexes, which with Zipf's law has the potential to be a good approximation of a regular web search**.

Chapter 3

ABS Model of the Atbrox Case Study

This chapter describes the resource-aware ABS Model for the ATB case study, how to orchestrate it and description of its initial use in continuous deployment. Figure 3.1 presents the actual cloud and mobile system (left) side-by-side with the ABS Model. The ABS model is end-to-end but does not cover newer parts of the architecture related to training deep learning networks.

3.1 Cloud Model - Sample from Open Data (M0)

English Wikipedia document samples - tokenized as list of words - are used as a model for crawled data in the ABS model. Wikipedia documents is a good choice to use as source data in the model because they are representative for most types of textual data and easy to scale up/down with in several directions: document size, number of languages, number of documents, multimedia documents. Wikipedia data can also be used to simulate periodic updates by either sampling from it or using actual Wikipedia updates.

Characteristics of Wikipedia documents are:

document size - average comparable to business documents (with outliers being book-sized)

number of languages - 288 languages

number of documents - Wikipedia has approximately 35 million pages

multimedia documents - not only text, but also multimedia content (images, video and sound)

3.1.1 ABS Model for Documents (M0)

The listing below shows an example of documents modelled in ABS as a list of individual tokens and bigram tokens that contains stop words. Documents are referred to with data type CrawledData in the ABS model.

```
List<Pair<URI, List<Word>>> documents =
    list[Pair("Agriculture in Canada",
              list["canada","canada is","is one","one","one of","of the","the largest","largest","agricultural", ..])];

data CrawledData = CrawledData(List<Pair<URI, List<Word>>> crawled_documents);
```

3.2 Cloud Model - Processing and Indexing with Mapreduce (M1 and M2)

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

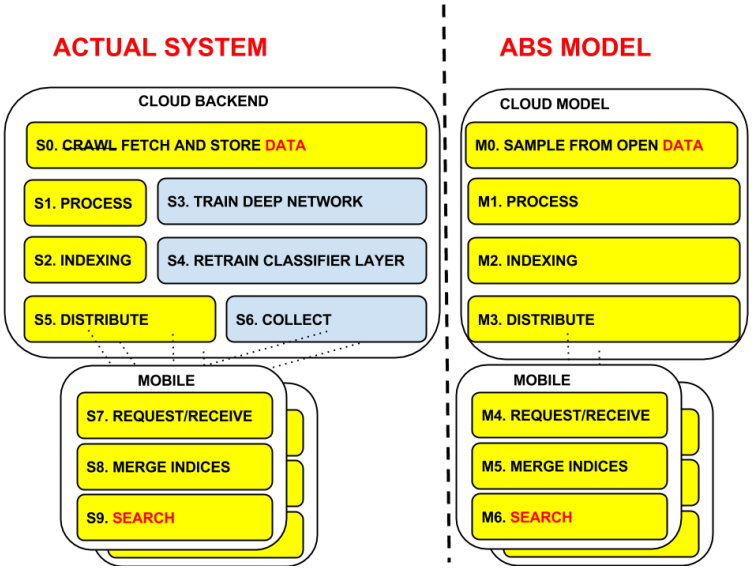

Figure 3.1: Actual System compared with ABS Model

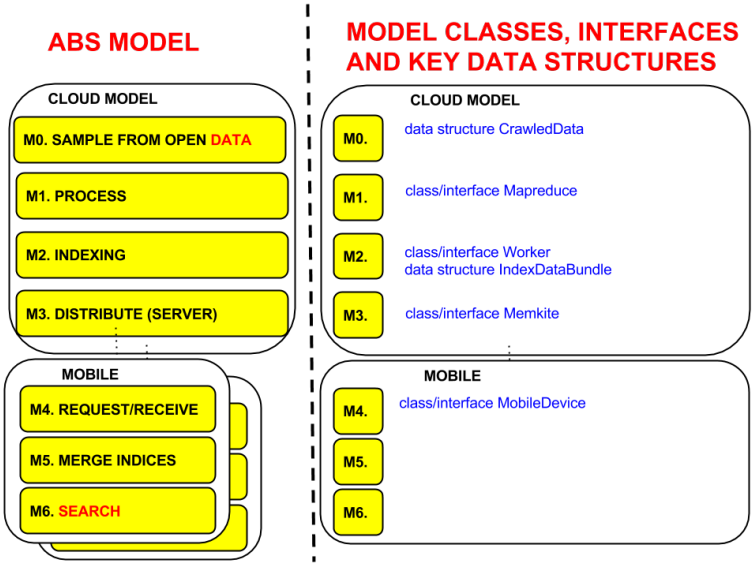

Figure 3.2: ABS model with classes, interfaces and data structures

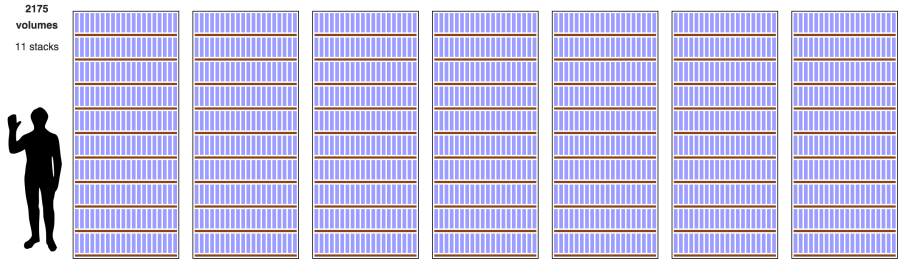

Figure 3.3: Size of English Wikipedia (Text Only) as 2175 volumes of 1.6M words each

index and the corresponding compressed content can be sent to the mobile device. Processing and indexing constitutes the intermediate stage in the cloud backend (Figure 2.2 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 by prefix index or inverted index. This involves indexing large amounts of data, and, in the case of prefix-indexing, heavy data processing. 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). In the ABS model we model processing with generic cloud-based parallel Mapreduce.

3.2.1 Relevant resources - Processing and Indexing

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:

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 map tasks = number of map 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.

In the ABS Mapreduce model for processing and indexing we are abstracting away from the HDFS file system, but instead model the number of virtual machine workers (and each of them as Deployment Components) in map and reduce processes as well as their resources specified in the cloud machine configuration - a given set of RAM, CPU, Bandwidth and Disk resources per virtual machine.

3.2.2 Functional Model - Processing and Indexing

We have modelled an inverted indexing job in ABS, reflecting how the Memkite system runs to prepare 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 MapReduce ABS model rely on two interfaces, one for the Worker and one for MapReduce:

Worker Interface

```
interface Worker extends DeployedObject {
  List<Pair<Word, URI>> invokeMap(URI key, List<Word> value);
  List<URI> invokeReduce(Word key, List<URI> value);
}
```

MapReduce Interface

```
// MapReduce
interface MapReduce {
    // invoked by client
    List<Pair<Word, List<URI>>> mapReduce(List<Pair<URI, List<Word>>>
        documents);
    // invoked by workers
    Unit finished(Worker w);
}
```

3.2.3 ABS Model of Cloud-deployed MapReduce (M1)

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.

CMapreduce class - uses a CloudProvider (e.g. a model of Amazon Web Services) and a set of machines with a given virtual machine resource configuration (similar to EC2 on AWS). It keeps track of a set of workers and assigns map and reduce tasks to them.

```
class CMapReduce(CloudProvider p, Int maxMachines, Map<Resourcetype, Int> machineConfiguration)
implements MapReduce
```

MapReduce Data *fMapResults* is used to store output data in the mapper, *intermediates* is used in the shuffler stage to group map results by key, *fReduceResults* is used for reducer output and *results* is the final result (e.g. an inverted index in this case study).

```
Set<Fut<List<Pair<Word, URI>>>> fMapResults = set[];
Map<Word, List<URI>> intermediates = map[];
Set<Pair<Word, Fut<List<URI>>>> fReduceResults = set[];
List<Pair<Word, List<URI>>> result = Nil;
```

Mapper phase - iterates through input data (documents - pair of URI and list of terms) and requests workers machines, then assigns data to the workers through async calls to the MapReduce map method (*invokeMap(key, value)*), and finalyy updates map results.

```
while (~isEmpty(items)) {
    Pair<URI, List<Word>> item = head(items);
    items = tail(items);
    Worker w = await this!getWorker();
    URI key = fst(item);
    List<Word> value = snd(item);
    Fut<List<Pair<Word, URI>>> fMap = w!invokeMap(key, value);
    fMapResults = insertElement(fMapResults, fMap);
}
```

Shuffler phase - When (intermediate) results produced by map workers are available it groups values together for each unique key that the map method assigned (i.e. dependent on what map() task it is solving)

```
while (~emptySet(fMapResults)) {
    Fut<List<Pair<Word, URI>>> fMapResult = take(fMapResults);
    fMapResults = remove(fMapResults, fMapResult);
    await fMapResult?;
    List<Pair<Word, URI>> mapResult = fMapResult.get;
    while (~isEmpty(mapResult)) {
        Pair<Word, URI> keyValuePair = head(mapResult);
        mapResult = tail(mapResult);
        List<URI> inter = lookupDefault(intermediates, fst(keyValuePair), Nil);
        intermediates = put(intermediates, fst(keyValuePair),
            Cons(snd(keyValuePair), inter));
    }
}
```

Reducer phase 1 - iterates through grouped-by-key intermediate data, requests worker machines and then assigns data to the workers - through async calls to the MapReduce reduce method (*invokeReduce(key, values)*)

```
Set<Word> keys = keys(intermediates);
while(~emptySet(keys)) {
  Word key = take(keys);
  keys = remove(keys, key);
  List<URI> values = lookupUnsafe(intermediates, key);
  Worker w = await this!getWorker();
  Fut<List<URI>> fReduce = w!invokeReduce(key, values);
  fReduceResults = insertElement(fReduceResults, Pair(key, fReduce));
}
```

Reducer phase 2 - collects reduce results produced by cloud workers.

```
while (~emptySet(fReduceResults)) {
  Pair<Word, Fut<List<URI>>> reduceResult = take(fReduceResults);
  fReduceResults = remove(fReduceResults, reduceResult);
  Word key = fst(reduceResult);
  Fut<List<URI>> fValues = snd(reduceResult);
  await fValues?;
  List<URI> values = fValues.get;
  result = Cons(Pair(key, values), result);
}
```

Modeling of Cloud Workers in MapReduce - launches a new virtual machine if needed and within resource limits or reuses an available virtual machine and returns back as a worker to the MapReduce framework.

```
Worker getWorker() {
  if (emptySet(workers)) {
      if (nWorkers < maxMachines) {
          DeploymentComponent machine = await p!launchInstance(machineConfiguration);
            [DC: machine] Worker w = new Worker(this);
            nWorkers = nWorkers + 1;
            workers = insertElement(workers, w);
        }
        await ~(emptySet(workers));
    }

   Worker result = take(workers);
     workers = remove(workers, result);
     return result;
}
```

3.2.4 ABS Model for Inverted Indexing with Mapreduce (M2)

Inverted indexing is the most common way of document indexing to support search; it provides fast lookup of a list of documents corresponding to a term.

Inverted Index Mapper Algorithm - algorithm that outputs pairs of word (term) and URI for each word in the document given as input.

```
Unit map(URI key, List<Word> value) {
        List<Word> wordlist = value;
        while (~(wordlist == Nil)) {
          [Cost:1] // cost annotation – simulates resource use
            Word word = head(wordlist);
            wordlist = tail(wordlist);
            this.emitMapResult(word, key);
        }
    }
```

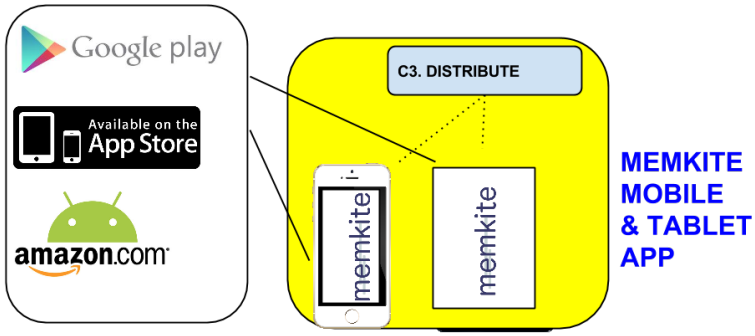

Figure 3.4: Distribution of Indexed Data to Mobile Customers

Wrapper method for Mapper Algorithm - this method communicates with Mapreduce framework and returns result back to it. This function is called from the Mapreduce worker virtual machine

```
List<Pair<Word, URI>> invokeMap(URI key, List<Word> value) {
    mapResults = Nil;
    this.map(key, value);
    master!finished(this); // tells the Mapreduce master that map call has been performed
    List<Pair<Word, URI>> result = mapResults;
    mapResults = Nil;
    return result;
}
```

Inverted Index Reduce Algorithm - algorithm that outputs all URIs for a given word (term).

```
Unit reduce(Word key, List<URI> value) {
    Set<URI> resultset = set(value); //removes duplicates
    while (~(emptySet(resultset))) {
      [Cost:1] // cost annotation – simulates resource use
        URI file = take(resultset);
        resultset = remove(resultset, file);
        this.emitReduceResult(file);
    }
}
```

Wrapper Method for Reducer Algorithm - this method communicates with the MapReduce framework and returns results back to it. This function is called from the MapReduce worker virtual machine.

```
List<String> invokeReduce(Word key, List<URI> value) {
    reduceResults = Nil;
    this.reduce(key, value);
    master!finished(this);
    List<URI> result = reduceResults;
    reduceResults = Nil;
    return result;
}
```

3.3 Cloud Model - Distribution of Indices to Mobile Device (M3)

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 3.3.

3.3.1 Relevant Resources - Distribution

A. Outbound bandwidth to transfer index elements to the app

B. CPU to schedule and perform index updates to apps (push) or react when mobile users request updates

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

The ABS model for distribution aligns directly with the described resources.

3.3.2 Functional Model - Distribution

We have modelled an distribution of indexed data from Cloud to Mobile Devices (Memkite Server) in ABS, reflecting how the Memkite system runs to send data for searching on the mobile client.

The distribution ABS model rely on one ABS interface with API that allows processing and indexing of data (process), and allows the mobile device to ask (in REST/RPC-manner) for an initial inverted index (fetchInitialConfiguration) and index updates (fetchNewData).

Memkite Interface

```
interface Memkite {
    Set<IndexDataBundle> process(CrawledData crawled_data);
    // Unit provision(Set<IndexDataBundle> new_data);
    Set<IndexDataBundle> fetchInitialConfiguration();
    Set<IndexDataBundle> fetchNewData(MobileDevice mobile_device);
}
```

3.3.3 Cloud-based ABS Model for Distribution and Serving (M3)

CMemkite class - uses a CloudProvider (e.g. a model of Amazon Web Services) and a set of machines with a given virtual machine resource configuration (similar to EC2 on AWS) to delegate resources to MapReduce processing.

```
class CMemkite(CloudProvider provider) implements Memkite
```

Memkite Data - *initial_configuration* is used for the initial index produced by MapReduce (*engine*); *known_mobile_devices* keeps track of all mobile devices that access the Memkite server; *provisioned_data* keeps track of index data that will be sent to mobile devices; *sent_data* keeps track of data that has been sent to mobile devices.

```
MapReduce engine;
Set<IndexDataBundle> initial_configuration = set[];
Set<MobileDevice> known_mobile_devices = set[];
Map<MobileDevice, Set<IndexDataBundle>> provisioned_data = map[];
Map<MobileDevice, Set<IndexDataBundle>> sent_data = map[];
```

process - starts MapReduce and creates indices for mobile devices.

```
Set<IndexDataBundle> process(CrawledData crawled_data) {
    List<Pair<URI, List<Word>>> documents = crawled_documents(crawled_data);
    List<Pair<Word, List<URI>>> results = await engine!mapReduce(documents);

    Set<IndexDataBundle> new_data = set[IndexDataBundle(documents, map(results))];
    initial_configuration = union(initial_configuration, new_data);
    while (~emptySet(new_data)) {
        IndexDataBundle bundle = take(new_data);
        new_data = remove(new_data, bundle);
        Set<MobileDevice> mobile_devices = keys(provisioned_data);
```

```
            while (~emptySet(mobile_devices)) {
                MobileDevice mobile_device = take(mobile_devices);
                mobile_devices = remove(mobile_devices, mobile_device);
                provisioned_data = put(provisioned_data, mobile_device,
                    insertElement(lookupUnsafe(provisioned_data, mobile_device), bundle));
            }
        }
        return set[];
    }
```

fetchInitialConfiguration - remote API called by mobile device to get initial index.

```
    Set<IndexDataBundle> fetchInitialConfiguration() {
        return initial_configuration;
    }
```

fetchNewData - remote API called by mobile device to get new data without any duplicates from what is already sent (Memkite server keeps track of what is sent to the device).

```
Set<IndexDataBundle> fetchNewData(MobileDevice mobile_device) {
  Set<IndexDataBundle> toSend = lookupDefault(provisioned_data, mobile_device, set[]);
  Set<IndexDataBundle> alreadySent = lookupDefault(sent_data, mobile_device, set[]);
  provisioned_data = put(provisioned_data, mobile_device, set[]);
  sent_data = put(sent_data, mobile_device, union(toSend, alreadySent));
  return difference(toSend, alreadySent);
}
```

3.4 Extended Cloud Model - Mobile Device (M4-M6)

From a user perspective the service is mainly the mobile device with an app that allows search on-device with fresh data, and from an ABS model perspective the mobile device can be seen as an extension of the cloud.

3.4.1 Relevant Resources - Mobile Device

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 Wi-Fi), 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 receive.

C. App and Index State: what data they have on the phone, possible index synchronization issues and index fragmentation (index merging may be needed 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.

3.4.2 Functional Model - Mobile Device

We have modelled the Mobile Device as having two methods: *pushButton* that fetches either the initial index or index updates from the cloud backend to mobile device, and *search* that searches for a particular query in the index on the mobile device.

MobileDevice Interface

```
interface MobileDevice {
  Unit pushButton();
  Set<URI> search(Term query);
}
```

3.4.3 Mobile Device ABS Model

CMobileDevice - objects of this class can run in parallel using Deployment Component (virtual machine) - with more limited resources (RAM, CPU and Disk) than the server configuration.

```
class CMobileDevice(Memkite memkite) implements MobileDevice
```

pushButton - checks if there is an index on the device; if it is not it fetches an initial index from the server, otherwise it fetches an index update and merges the existing index with the index update. Note that this is an operation that is easy to do in the ABS model but very hard on the device, this because merging of indices is costly resource-wise (RAM, CPU and disk operations on device) and may be interrupted if it is running in the background since mobile operating systems (like iOS) have resource-restrictions when running in the background.

```
Unit pushButton() {
    if(emptySet(local_data)) {
        local_data = await memkite!fetchInitialConfiguration();
  } else {
      Set<IndexDataBundle> new_data = await memkite!fetchNewData(this);
      local_data = this.merge_data(local_data, new_data);
  }
}
```

search - searches the index on the mobile device and returns a list of results.

```
Set<URI> search(Term query) {
  Set<IndexDataBundle> to_search = local_data;
  Set<URI> results = set[];
  while(~emptySet(to_search)) {
    IndexDataBundle index_data_bundle = take(to_search);
    to_search = remove(to_search, index_data_bundle);
    InvertedIndex inverted_index = inverted_index(index_data_bundle);
    PostingList posting_list = lookupDefault(inverted_index, query, list[]);
    while(~isEmpty(posting_list)) {
      URI uri = head(posting_list);
      posting_list = tail(posting_list);
      results = insertElement(results, uri);
    }
  }
  return results;
  }
}
```

3.5 Cloud Orchestration of ABS Model

This shows an example of cloud orchestrating 2 mobile devices (with CPU resource of 5) and 1 Memkite server (with CPU resource of 20) on a simulated Amazon Web Service cloud.

Setting up Cloud Backend and two Mobile Devices

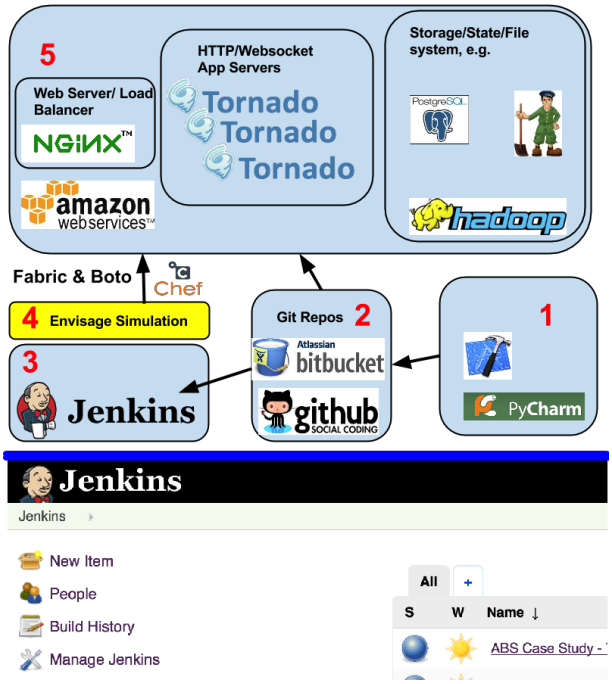

Figure 3.5: Initial use of ATB ABS model with Jenkins continuous deployment

```
CloudProvider provider = new CloudProvider("Amazon");

DeploymentComponent dc = new DeploymentComponent("Memkite", map[Pair(CPU, 20)]);
[DC: dc] Memkite memkite = new CMemkite(provider);

DeploymentComponent mobile_device_dc = new DeploymentComponent("MobileDevice", map[Pair(CPU, 5)]);
[DC: mobile_device_dc] MobileDevice mobile_device = new CMobileDevice(memkite);

DeploymentComponent mobile_device_dc2 = new DeploymentComponent("MobileDevice2", map[Pair(CPU, 5)]);
[DC: mobile_device_dc2] MobileDevice mobile_device2 = new CMobileDevice(memkite);
```

Note that the Memkite server does the cloud orchestrating of MapReduce; here is an example of using (up to - depending on amount of input data) 5 virtual machines, each with CPU capacity of 20 and memory capacity of 10000:

```
engine = new MapReduce(provider, 5, map[Pair(CPU, 20), Pair(Memory, 10000)]);
```

Start processing of crawled data with Hadoop/MapReduce on Cloud

```
await memkite!process(CrawledData(documents));
```

Get data onto mobile devices and search

```
await mobile_device!pushButton();
await mobile_device2!pushButton();

Set<URI> search_results_on_mobile1 = await mobile_device1!search("in");
Set<URI> search_results_on_mobile2 = await mobile_device2!search("in");
```

3.6 Current use of ATB ABS Model with Continuous Deployment

The current ATB ABS Model is integrated with continuous deployment so it covers step 1-4 (but not 5) as shown in the upper part of figure 3.5. What runs in Jenkins (the continuous deployment framework) now is

basically a system test for a resource-aware model, meaning that it runs through entire model.

Chapter 4

Conclusions

This deliverable has presented our work on an end-to-end resource-aware model for the 7 parts (M0-M6) of the ATB case study (figure 3.2):

M0 Sample from Open Data - What data to make searchable on the Mobile Device

M1 Processing of Data - Model of the Parallel Mapreduce Framework

M2 Indexing of Data - Model of Mapreduce-based Inverted Indexing Algorithm

M3 Distribute Data to Mobile Devices - Modeling of bandwidth

M4 Request and Receive Data from Cloud to Mobile Device - Modeling of REST-like API that can be called from Mobile Device

M5 Merge Indices on Mobile Devices - Modeling what needs to be done to handle index updates from Cloud Backend

M6 Search on Mobile Device - Models the primary purpose of the application

Challenges ahead are evaluating the model compared to the real deployment and tuning resource parameters (e.g. CPU, RAM, Bandwidth and Number of virtual machines) in addition to parameters controlling resources consumed (cost-annotations) so it matches the real deployment well. When we achieve that we can change the model prior to changing the actual deployment in order to estimate resource use; at that level the model is ready to become a part of complete continuous deployment. When in Continous Deployment it can also be extended with other Envisage outcomes related to automated unit test creation from the ABS Model and deadlock detection. In addition to simulated costs from running the ABS model we can also get a more careful and conservative cost-estimates using the static cost analysis tools developed in Envisage. We also need to iterate on the scale of simulations with the ABS model (e.g. increase the number of concurrent mobile devices connecting to the server and the resources made available on the cloud side).

4.1 Experience and Feedback to Technical Tasks

In this wection, we comment on our experiences gained from the ATB case study with respect to the technical tasks of Envisage.

Task T1.2: Modeling of Resources Resources modelled in ABS that are used in the ATB case study include:

Cloud Computing Level - API for access to several Cloud Vendors, e.g. Amazon Web Services, where virtual machines can be requested

Virtual Machine Level - API for specifying resources on individual machines, e.g. RAM, CPU and Disk resources

The support for modeling cloud resources with ABS is overall good, but given the increasing importance of mobile becoming the primary point of contact between people and the cloud, abstractions for modeling mobile resources could be an advantage to add to ABS.

Task T1.3: Modeling of Deployment In the ABS model we cover 2 different types of deployment:

1. Cloud Backend - Using the CloudProvider class (set to Amazon Web Services) and the DeploymentComponent class for each virtual machine started.

2. Mobile Devices - Also using the CloudProvider and DeploymentComponent, but reducing the amount of resources per virtual machine instance since they represent mobile phones and not cloud virtual machines in data centers. This modeling aligns well with Amazon's recent announcement of the AWS Device Farm[1] - where one can deploy to physical mobile hardware through a cloud API. One difference is that the AWS Device farm is primarily for testing on various mobile hardware configurations (e.g. choose from several vendors of Android phones) rather than (ABS simulation of) testing on scale - which is more of interest in the case study.

In actual cloud deployment there is typically a separation between 1) provisioning of resources and 2) performing software/data deployment on provisioned resources, where the prior might be specific for a cloud vendor (e.g. AWS has tools like Elastic Beanstalk[2] or third party libraries like Boto[3] for doing this - ATB uses Boto), and the second is usually more generic and is similar for most IAAS clouds (e.g. using tools like Chef[4] or Puppet[5] - ATB uses Chef). The provisioning support in ABS is good (see also T1.2), but support for modeling of (automated) software/data deployment could be extended. Typical issues that needs to be handled in software/data deployment is version conflicts (e.g. on library dependencies), configuration file handling, security (e.g. creation and distribution of encryption keys and SSL certificates) and tight integration with continuous deployment system.

Task T1.4: Simulation The model supports simulation end-to-end, from the data arrives in the cloud service until it has become searchable on the mobile devices. It also supports simulation with varying amounts of load (e.g. amounts of data to index and number of mobile devices) and varying resources to handle the load (e.g. number and capacity of virtual machines used for indexing and serving of data in the cloud backend).

Since individual latency - from the data arrives in the cloud until it arrives on the mobile device - is of key importance for the SLA, a useful addition to the model could be to add "tracer bullet" style logging that follows the data through the system, i.e. annotate data with timestamps in every step of the model in order to see where it uses time.

Task T2.2: Service Contracts and SLAs The primary SLA for ATB is how fast data that has arrived in the Cloud Backend becomes searchable on a customer's mobile device. The model, which allows simulations of that time, is given various scenarios relevant for the SLA, e.g. mobile bandwidth capacity for customers. Since latency is a common metric in SLAs, we expect that the support for SLAs can be combined with the suggested "tracer bullet" approach for simulation in T1.4.

[1] http://aws.amazon.com/device-farm/
[2] http://aws.amazon.com/documentation/elastic-beanstalk/
[3] https://boto.readthedocs.org
[4] https://www.chef.io
[5] https://puppetlabs.com/

Task T2.3: Monitoring Add-ons This combined with tuning of the model is where the main remaining work is currently expected, i.e. monitor if SLA is fulfilled and visualize the state of the mobile devices during simulations (e.g. amount of index data received and mobile bandwidth).

Appendix A - Feasibility of Large-Scale Mobile Offline Search

Mobile devices are approaching recent supercomputers in capacity (see figure 1.4 comparing web search with a tablet) and this might change how people will use them for search.

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), 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 2), 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.

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.

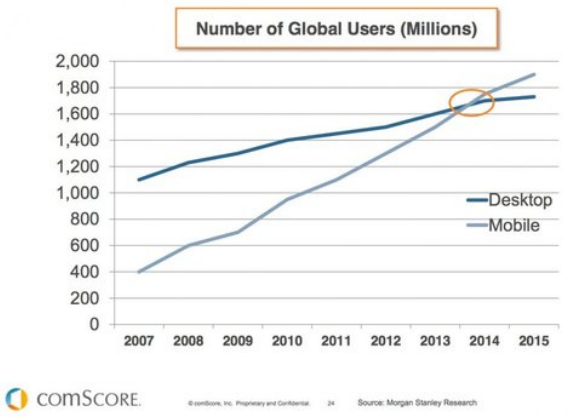

Figure 1: Mobile surpasses Desktop

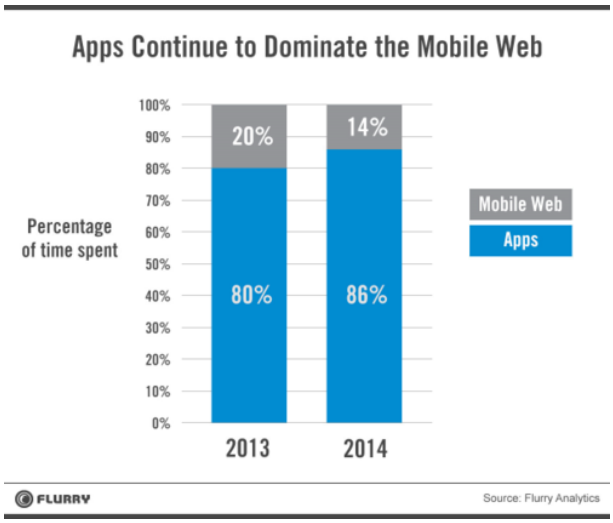

Figure 2: Apps replacing Mobile Web

Figure 3: iPhone faster than Cray Supercomputer, Tweet Posting March 16, 2014

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 3). 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, 4]. 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 [6]:

1. Qualcomm announced mobile 64 bit CPUs Snapdragon 808 and 810
2. Intel announced the 22 nanometer 64 bit Atom CPU
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 2.3.1 discuss the experiences gathered by ATB concerning large-scale inverted index on an iPad Mini.

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

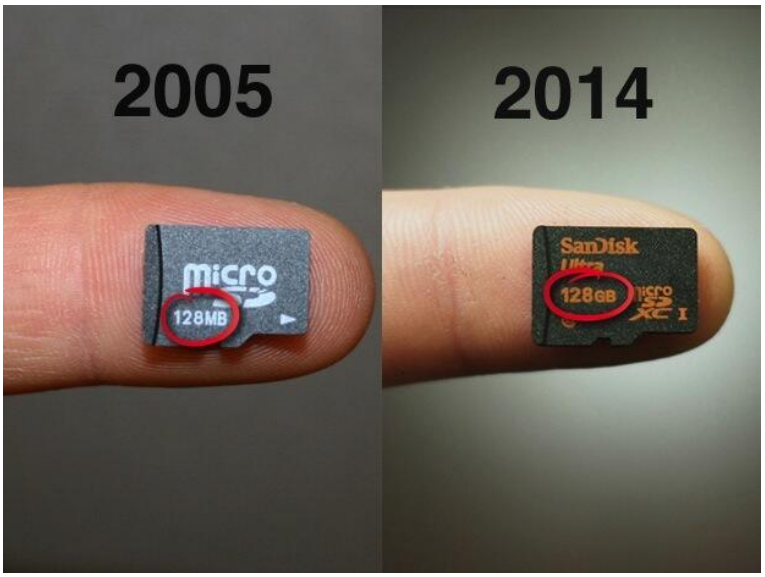

Figure 4: 1000-fold increase in storage on MicroSD cards in 9 years

Figure 5: More data on USB stick in 2014 than Web Search Cluster in 1998/99

from 128 MB to 128 GB (Figure 4), 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, 13], 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 5).

While Flash/SSD-based storage used in MicroSD cards provides large mobile storage (approximately 700 GB per cubic cm), two technologies can replace it [10]:

1. RRAM: potential of storing 1 terabyte per square cm chip, with up to 20 times faster retrieval rates than SSD [9]

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 [7]

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.

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-hasw March, 2014.

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

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

[14] 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.