REPORT ON DELIVERABLE 5.1.1

Big Water Data Management engine

PROJECT NUMBER: 619186
START DATE OF PROJECT: 01/03/2014
DURATION: 42 months

DAIAD is a research project funded by European Commission's 7th Framework Programme.

Dissemination Level	Public
Due Date of Deliverable	Month 12, 28/02/2015
Actual Submission Date	25/02/2015
Work Package	WP5 Big Water Data Analysis
Task	Task 5.1 Big Water Data Management
Type	Prototype
Approval Status	Submitted for approval
Version	1.0
Number of Pages	16
Filename	D5.1.1_DAIAD_Big_Water_Data_Management_engine.pdf

Abstract

This report presents the design and implementation of the *Prototype Deliverable D5.1.1 "Big Water Data Management engine"* which will be used for storing and processing water consumption data in the context of the DAIAD project. The document outlines the architecture of the big data management engine, describes the rationale behind the selection of the NoSQL database which is an integral part of the architecture, and enumerates several database schema design approaches for storing and querying water consumption time series data. The prototype has been deployed on DAIAD's cloud infrastructure and has been integrated in the early DAIAD integrated system (MS4 milestone: DAIAD@feel, DAIAD@know, DAIAD@home), receiving, managing and supporting analysis queries for water consumption data.

History

version	date	reason	revised by
0.1	8/01/2015	First draft	Yannis Kouvaras
0.2	15/01/2015	Revisions and improvements in various Sections	Yannis Kouvaras
0.3	20/01/2015	Revisions and improvements in various Sections	Giorgos Giannopoulos
0.4	26/01/2015	Minor edits, addition of figures	Yannis Kouvaras
0.6	10/02/2015	Changes in Section 2	Yannis Kouvaras
0.8	17/02/2015	Revisions and improvements in various Sections	Sophie Karagiorgou
1.0	25/02/2015	Final version	Spiros Athanasiou

Author list

organization	name	contact information
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Yannis Kouvaras	jkouvar@imis.athena-innovation.gr
ATHENA RC	Sophie Karagiorgou	karagior@imis.athena-innovation.gr
ATHENA RC	Giorgos Giannopoulos	giann@imis.athena-innovation.gr

Executive Summary

This report presents the design and implementation of the *Prototype Deliverable D5.1.1 "Big Water Data Management engine"*. The Big Water Data Management engine is the infrastructure which will host the components of the Big Data analysis services for the complete DAIAD system.

Within this scope, we outline the architecture of the big data management engine, identify the software platforms used in our deployment and present the undertaken implementation setup. Particularly, we concentrate on the cloud infrastructure which is built in the form of a private Infrastructure-as-a-Service (IaaS) cloud by deploying a MapReduce (Apache Hadoop, Flink) framework.

Further, we present the YARN resource manager which supports resource management functionality to arbitrary data processing models.

In order to effectively contend data storage and queries we suggest an architecture which supports the HBase NoSQL database and relevant database schema design approaches for water consumption time series data.

The proposed schemas for data analysis form the basis for the subsequent work to be carried out, but are also subject to modifications and adjustments, if needed, depending on the data analysis requirements that will emerge during the progression of the project.

Abbreviations and Acronyms

HDFS	Hadoop Distributed File System
SSH	Secure Shell
WAL	Write-Ahead Log
IaaS	Infrastructure as a Service
YARN	Yet Another Resource Negotiator
WAL	Write-Ahead Log

Table of Contents

1. Introduction

This report presents the design and implementation of the Big Water Data Management engine. The infrastructure is built in the form of a private Infrastructure-as-a-Service (IaaS) cloud and will be used for storing and processing large scale water consumption data as described in Task 5.1.

The remainder of this document is structured as follows.

In Section 2, we revisit the architecture described in deliverable D1.2 "DAIAD Requirements and Architecture", enumerate the software platforms that we are using in our deployment and present the current implementation setup.

In Section 3, we discuss the features that influence the determination of the NoSQL database which is incorporated in our solution.

Finally, in Section 4 we present a preliminary analysis of data access patterns for different workload types and discuss various schema design options.

2. Architecture

In deliverable D1.2 we identified two distinct query workload types, namely, *real-time low latency queries* and *high-latency analytical queries*. In order to enhance performance for both types we had suggested the implementation of workload isolation by installing and configuring two separate clusters. Data will be exchanged by the two clusters using both online replication and batch data handling and processing jobs.

In Figure 1 we depict the stack of software projects that we used in DAIAD architecture. At the lower level is the Hadoop Distributed File System (HDFS)[1] which offers reliable and redundant storage for the components which are lying higher in the software stack. HDFS supports high throughput data access of large datasets and is not optimized for low latency.

On top of HDFS, YARN[2] resource manager and HBase[3] NoSQL database are located. YARN is a generic resource manager that offers resource management functionality to arbitrary data processing models. HBase is a NoSQL database that can handle *high volume, high velocity real-time data*. HBase also supports low latency, random data access over HDFS.

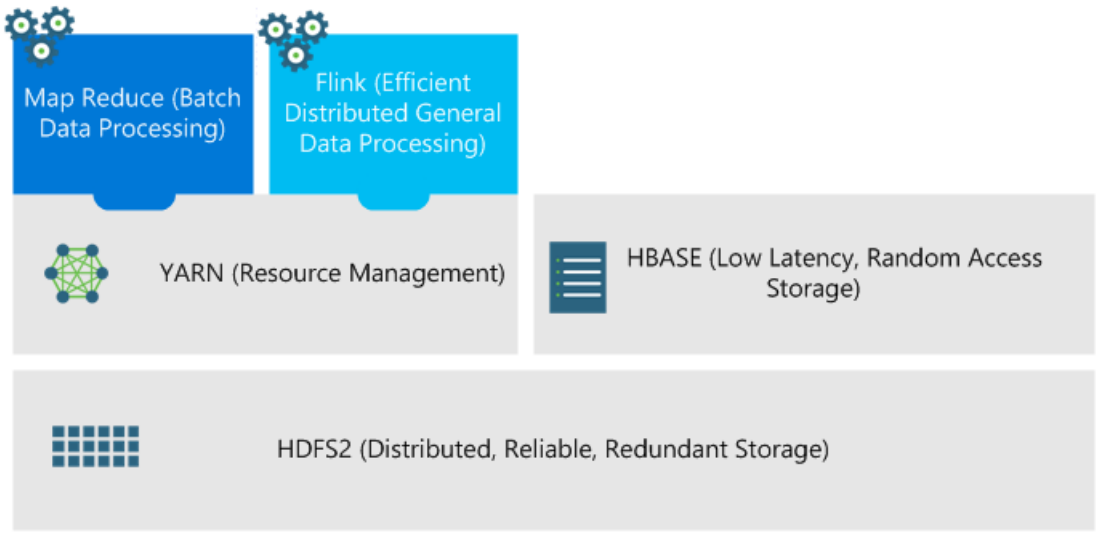

Figure 1: DAIAD Big Data Engine Software Stack

Finally, the Hadoop MapReduce and Flink[4] data processing frameworks are built at the top of the software stack. Both frameworks utilize YARN for controlling resource management[5] and can use HDFS and HBase both as a *data source* and a *data sink*. Hadoop MapReduce is optimized for batch processing of large

[1] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[2] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[3] http://hbase.apache.org/
[4] https://flink.apache.org/
[5] Hadoop MapReduce and Flink can operate independently from YARN but the latter offers better cluster resource control and utilization.

datasets, while Flink is a more general purpose data processing framework with powerful characteristics for iterative processing. Moreover, Flink supports cost based optimization for data operations reordering and offers compatibility with existing Hadoop MapReduce operators. The two latter features make Flink a candidate for completely removing Hadoop MapReduce from our software stack. Nevertheless, Hadoop MapReduce will still be available through the HDFS and YARN installation package.

Having an overview of the software stack, we will continue with the presentation of the physical configuration of the Big Data Engine cluster and the description of processes executed by each host in it. Figure 2 depicts the type of hosts in the cluster, their cardinality and the processes which can be executed over them.

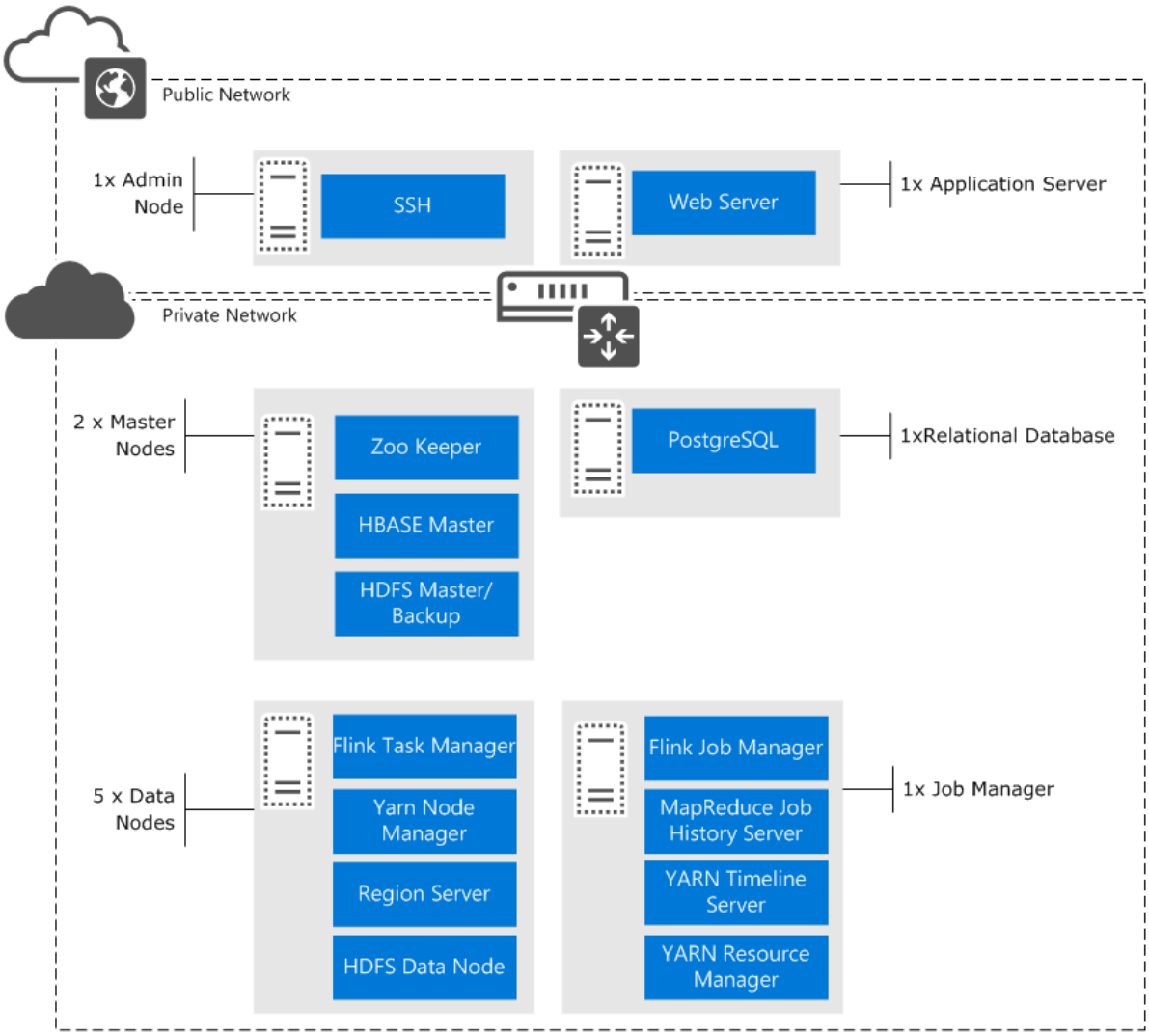

Figure 2: DAIAD Big Data Engine cluster

In the following we give a more detailed description of the included components:

- *5xData Nodes*. Each data node is responsible for storing data. Every data node executes an HDFS DataNode process for handling HDFS data, an HBase RegionServer for managing table regions, and

a YARN Node Manager for handling execution of job tasks. Depending on the storage and computational requirements more data nodes may be added.

- *1xJob Manager*. A single job manager server is responsible for handling analytical job submission for both Hadoop MapReduce and Flink data processing frameworks. Moreover, it hosts the processes of YARN Timeline Server and MapReduce Job History Server that provides current and historic information about running and completed applications.
- *2xMaster Nodes*. A master node hosts an HDFS NameNode (master or backup), an HBase Master and a ZooKeeper instance. If additional redundancy and fault tolerance is required, more master nodes may be added.
- *1xRelational Database*. A single relational database node hosting a PostgreSQL instance for storing smaller datasets is required for application and cluster administration.
- *1xAdmin Node*. A single administration data node that acts (a) as a gateway for accessing the cluster private network and administrating hosts remotely using secure shell (SSH) and (b) as a proxy for the HDFS, HBase and Flink administration sites.
- *1xApplication Server*. The application server will host all the web applications implemented by DAIAD such as DAIAD@home, DAIAD@commons and DAIAD@utility. More application servers may be added along with a load balancer if capacity issues emerge.

At the current state of implementation, we have configured a single cluster of virtual servers with the whole software stack installed for handling both workload types, i.e. real-time and analytical queries. Such a configuration will suffice for development, testing and benchmarking. During the implementation of Task 5.1.3 “Updated Big Water Data Management engine”, a second cluster will be installed and data replication processes will be configured. More precisely, for replication we intend to apply the following methods:

- For real-time data, we are going to synchronize the HBase databases using the Write-Ahead Log (WAL). Updates will be propagated from the cluster handling real-time data to the data analysis cluster.
- For data aggregation, the data analysis cluster will compute the aggregates using batch jobs and will store the results directly to the real-time cluster.

3. NoSQL Database

One of the most critical decisions during the design of the Big Water Data Management engine was the selection of the NoSQL database that would store the water consumption data. The two most prevalent candidates were Apache HBase and Apache Cassandra. Both, HBase and Cassandra, are wide column key value stores that can handle high volumes and high velocity data. We decided to use HBase mostly because of the following features:

- *Powerful Consistency*. HBase supports powerful consistency. Since all read and write operations are always routed through a single RegionServer, it is guaranteed that all writes are executed in order that all reads retrieve the most recent committed data.
- *Hadoop/HDFS Integration*. HBase is fully integrated with HDFS which makes it a good option for batch analytics using Hadoop MapReduce. Moreover, Flink already offers support for reading data from both HDFS and HBase.

Nevertheless, HBase has a few shortcomings with the most important described next:

- *Complex Configuration*. In general HBase has too many moving parts in comparison to the Cassandra master-less architecture. Apart from the RegionServers, at least one HBase Master and one ZooKeeper are required with each of them being a single point of failure.
- *RegionServer Recovery Time*. Since every region is managed by a single RegionServer, when a RegionServer crashes, its data is unavailable until a new RegionServer takes control of it. Recovery process may take several minutes since edits should be replayed by the WAL. Nevertheless, latest HBase releases have decreased this downtime significantly.

Despite the aforementioned disadvantages, we have opted for HBase since we had already selected HDFS as our distributed storage. HDFS can be also used as a data source and as a data sink for task analytics in the analytics processing cluster. Using Cassandra would require extra persistence redundancy. Moreover, using a different NoSQL database per cluster would have complicated replication between clusters and increased administration effort.

4. Data Schema

In this section, we present a preliminary design of the database schema used for storing water consumption data. Due to the *data-centric* nature of wide column key value stores such as HBase, the two most important decisions affecting performance when designing a schema are the *form of the row keys* and *data storage redundancy*. In order to make well-informed choices we have to identify data access patterns required by the application use cases first. In the next two sections, we examine solutions for real-time user queries and analysis queries where low latency is not always required.

4.1. Real-time Queries

In order to categorize common data access patterns, we enumerate a few common queries expected by the DAIAD@home and DAIAD@commons applications:

- A user may request information about the water consumption over a specific time interval. The query may include more than one measurement values i.e. volume along with temperature. Moreover, the user may have more than one registered devices. In this case, either both time series may be rendered separately or data may be aggregated and presented as single time series.
- The user may request information about the water consumption over a specific time interval using variable temporal granularity i.e. hourly, daily, monthly etc.
- Similar to queries 1 and 2, a user may query the average water consumption of a DAIAD@commons community he belongs to. It is also possible to select data from more than one communities.

There are several *schema-design* approaches for answering efficiently the aforementioned queries. Our goals are to (a) achieve even data distribution across all RegionServers, (b) minimize the index size in comparison to the actual data and (c) preserve user data locality. The latter is essential in order to accelerate range scans since users will probably request data over a time interval instead of just a single point in time.

Starting with the row key schema, we have to decide what information and in which order will be stored in the row key. Since users most of the time request data for themselves, user unique identification key (Id) is a good candidate for the row key. Moreover, a good practice is to hash the Id value in order to achieve better data distribution across RegionServers. In addition, hashing guarantees that the user Id hash is of fixed size which simplifies row key management during development.

Next, users may request data per device, thus, we can concatenate the user hash with the device unique Id. Again, we can use a hash of the device unique Id. Nevertheless, the number of devices per user is expected to be small. Therefore, instead of creating a hash, we will store associations between user and device to a different table and will assign a two bytes code to each association. This alternative design should also be tested using sample data to verify if it affects row distribution across RegionServers since the cardinality of this field is expected to be small.

Finally, we append the measurement timestamp to the row key in order to preserve ordering. In addition, since the users may prefer to view the most recent measurements first, we may invert time ordering by storing the difference between the maximum long value and the timestamp.

After designing the row key, the column families and column qualifiers must be declared. In general, HBase documentation suggests using short column family names since they are stored as part of every key value cell. Using short column family names decreases both storage space and network traffic. Moreover, it is encouraged not using many family columns. In our scenario we will be using a single family column, named "m".

With the current row key selection each row should store a single measurement value. This type of schema design is usually referred to as a *"tall" schema*. By modifying the timestamp as part of the row key we can store timestamp as a column qualifier instead of as a part of the row key, thus having stored all measurements for a specific device and user in a single row, resulting in a "wide" schema. Moreover, we can store a prefix of the timestamp as part of the row key and the rest of it as a column qualifier, creating a hybrid schema where each row stores multiple measurements for a single device over a long time interval.

The latter approach is similar to the technique used by OpenTSDB[6] and is the most promising approach since it offers a *configurable query performance*. OpenTSDB is a time series database built on top of HBase. Although, we could try and adapt OpenTSDB to our needs, introducing a new layer of services over the existing architecture does not simplify the effort. Figure 3 depicts the form of the row key using a hybrid schema like OpenTSDB. Timestamp prefix length may vary based on the size of the time interval each row should represent. For our scenario that targets households, hourly intervals will probably suffice.

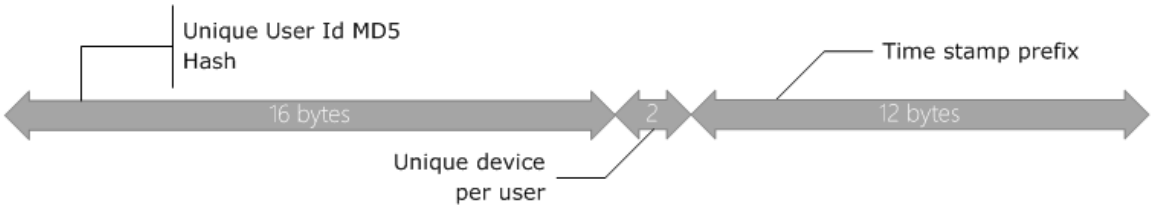

Figure 3: Row Key

Finally, using the hybrid schema, measurements will be saved using column qualifiers consisting of the timestamp offset and the name of the measurement. In contrast to OpenTSDB which stores a single metric measurement per cell, we store all measurements for a single timestamp since we expect users to request more than one measurements at the same time. Moreover, although the Amphiro device returns a compact representation of measurements, as depicted in Figure 4, we decided to create a single column qualifier per measurement in order to accommodate for different metrics or devices i.e. smart water meters.

[6] http://opentsdb.net/ The Scalable Time Series Database

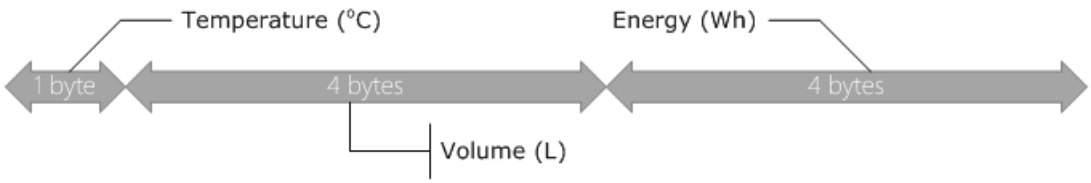

Figure 4: Amphiro data output

In Figure 5, a single cell with two different measurements at timestamps t1 and t2 is depicted.

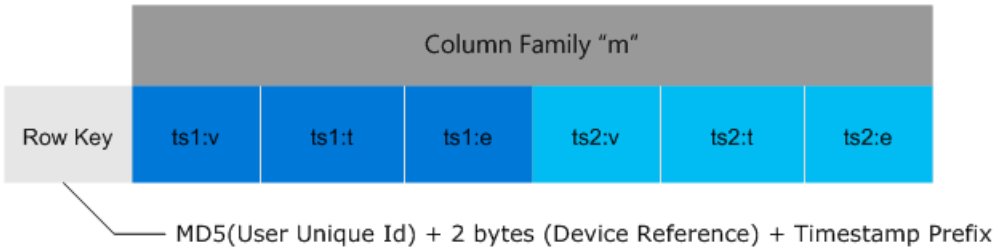

Figure 5: HBase Cell Value with two measurements

Using the schema shown in Figure 5, query 1 can be easily answered using short range scans. Likewise, query 2 can be answered but processing will become less efficient as the time interval size increases. However, query 3 cannot be executed efficiently using such a schema since it will require at least one scan operation for every member of a DAIAD@commons community. We may solve this problem using two different approaches. We will either use a batch job that will periodically aggregate community data and store the results in HBase or will create a new table that will store redundant copies of measurements of all community users every time a new measurement is inserted for any of them. Such a table will have a similar schema like the user table but instead of the user unique Id hash, the community unique Id hash will be stored in the row key. Moreover, the device Id will be removed too. For query 3, since a valid assumption is that community members will probably require data at half an hour, hourly or even larger time intervals, the aggregation solution seems better. Using a second table for storing community measurements will accelerate the batch job execution but will also increase write workload, especially if many users are subscribed to many communities.

4.2. Analysis Queries

In the previous section we have presented schema design solutions for queries mostly expected by DAIAD@home and DAIAD@commons applications. In this section, we examine several data schemas focused on high-latency data analysis queries.

In general, user applications tend to submit user-oriented queries. On the contrary, in data analysis applications experts are more likely to query data over time for a disparate set of users attempting to extract useful information about user behavior or to detect time series patterns with correlation to specific user characteristics i.e. user age, education etc. Examples of such queries are the following:

- Find the average daily water consumption over the last three months for all users.
- Find the average daily water consumption over the last three months for several age brackets.
- Compute the average weekly water consumption over the last year for users with a specific place of residence. Such queries can become more complex if residence address is not determined by city name or postal code but by the coordinates of a polygon.

In every case, the real-time data will always be replicated to the analysis cluster using the schema presented in the previous section. Such a schema will allow data analysts to have access to water consumption data at its finest granularity. Still, such a schema cannot be used for answering queries like the aforementioned ones. Since most of the queries refer to all users and place constraints over the time dimension, the implemented row key design does not allow efficient range scans. Consequently, a full table scan is required which is inefficient if *petabytes of data must be scanned*. At the same time, storing data using the timestamp as the prefix of the row key is also inefficient and will lead to unbalanced utilization of the cluster nodes. In particular, the ever increasing timestamp prefix value will always cause a single RegionServer accepting all insert operations even if data regions have been split between multiple RegionServers.

A possible solution for balancing data distribution is to use a *salt* as a prefix when creating a row key and append the timestamp. For instance, using the remainder of division (*modulo operation*) of the timestamp by the number of RegionServers as the row key prefix, will create a more balanced row distribution. Still, answering a query over a time interval will require executing scans over all servers. Another solution is trading time granularity over dataset size by using pre-aggregation. Full table scans will still be required but the size of data will be several times smaller.

The same problem, i.e. inefficient range scans, occurs for queries that filter data over specific row attributes such as query 2. HBase has no support for joins. Thus querying data from several tables may require either creating copies of the data with the appropriate columns and row keys holding all the required information, or implementing joins manually in code i.e. loading if possible all users in memory and performing a hash join.

Likewise, as with the temporal analysis problem, spatial analysis suffers from optimizing range scans. In spatial analysis the main problem is to preserve spatial locality when distributing data across the RegionServers. Different techniques can be applied like (a) using z-ordering or GeoHash[7] for prefixing the row key or (b) implementing more sophisticated models like HGrid[8] . In any other case redundant storage is required.

[7] http://en.wikipedia.org/wiki/Geohash

[8] http://dl.acm.org/citation.cfm?id=2515067 HGrid: A Data Model for Large Geospatial Data Sets in HBase

Finally, for ad-hoc analysis queries for which users expect low latency, we are going to employ pre-aggregation on different levels over the time dimension. Hence initial data with time granularity of seconds will be aggregated per user and device over half an hour, hourly, daily and weekly intervals.

In conclusion, we have to notice that the suggested schemas for data analysis are likely to significantly change depending on the data analysis requirements that will emerge during the progression of Tasks 5.2, 5.3 and 5.4.