REPORT ON DELIVERABLE 5.1.2

Updated 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 36, 28/02/2016
Actual Submission Date	28/02/2016
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	74
Filename	D5.1.2_Updated_DAIAD_Big_Water_Data_Management_engine.pdf

Abstract

This document presents an overview of the Prototype Deliverable D5.1.2 "Updated Big Water Data Management Engine", which includes the *final* DAIAD software components for managing, processing, and disposing all of DAIAD's data (*e.g., water consumption time-series, household metadata, system settings*). First, we present a high-level overview of the DAIAD Big Data Management engine, its various components, purpose of use, and characteristics. Then, we present the deployment of the engine in our private IaaS (Infrastructure-as-a-Service) cloud. In the following, we detail the database schemas applied, cover their evolution throughout the course of the project, and all optimizations applied to increase scalability and performance. Finally, we present all ETL facilities for importing, cleaning, and exporting DAIAD data, and the Data API, which exposes data management and processing services across all sub-systems and components of the complete DAIAD system.

History

version	date	reason	revised by
0.1	10/01/2016	First draft	Yannis Kouvaras
0.2	15/01/2016	Revisions and improvements in various Sections	Spiros Athanasiou
0.3	18/01/2016	Revisions Data API sub-section	Nikos Georgomanolis, Stelios Manousopoulos
0.4	25/01/2016	Detailed schemas and Annexes	Yannis Kouvaras
0.5	02/02/2016	Revisions and improvements in various sections	Giorgos Giannopoulos
0.6	04/02/2016	Updated introduction and executive summary	Spiros Athanasiou
0.7	10/02/2016	Revisions and improvements in various Sections	Stelios Manousopoulos
0.8	15/02/2016	Revised and completed Annexes	Yannis Kouvaras
0.9	22/02/2016	Revisions and improvements in various Sections	Nikos Karagiannakis
1.0	28/02/2016	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	Giorgos Giannopoulos	giann@imis.athena-innovation.gr
ATHENA RC	Nikos Georgomanolis	ngeor@imis.athena-innovation.gr
ATHENA RC	Stelios Manousopoulos	smanousopoulos@imis.athena-innovation.gr
ATHENA RC	Nikos Karagiannakis	nkara@imis.athena-innovation.gr

Executive Summary

This document presents an overview of the Prototype Deliverable D5.1.2 "Updated Big Water Data Management Engine", which includes the *final* DAIAD software components for managing, processing, and disposing all of DAIAD's data (*e.g., water consumption time-series, household metadata, system settings*). DAIAD's Big Data engine has been tested and validated in the context of our 12-month real-world trials, successfully addressing all scalability, performance, and fault-tolerance requirements. Throughout the course of the project, several *major and minor revisions* have been performed in the Prototype Big Data Engine (D5.1.1, M12) in response to (a) evolving requirements across all of DAIAD's system aspects (*e.g., from mobile applications, to analysis services for demand experts*), (b) changing structure, content, and quality of DAIAD's data sources (*e.g., household metadata, low quality of SWM data*).

In Section 1, we present a high-level overview of the DAIAD Big Data Management engine, its various components, purpose of use, and characteristics.

In Section 2, we present the deployment and integration of the engine in our private IaaS (Infrastructure-as-a-Service) cloud.

In Section 3, we detail the database schemas applied, cover their evolution throughout the course of the project, and all optimizations applied to increase scalability and fault-tolerance.

In Section 4, we present all ETL facilities for importing, cleaning, and exporting DAIAD data, and the Data API, which exposes data management and processing services across all sub-systems and components of the complete DAIAD system.

Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
BT	Bluetooth
CSV	Comma Separated Values
DOM	Document Object Model
DTO	Data Transfer Object
ETL	Extract, Transform and Load
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
Java EE	Java Enterprise Edition
JSON	JavaScript Object Notation
MR	MapReduce
REST	Representational State Transfer
SFTP	SSH File Transfer Protocol
SQL	Structured Query Language
SSH	Secure SHell
SSH2	Secure SHell 2
SWM	Smart Water Meter
TCP/IP	Transmission Control Protocol/Internet Protocol
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
VM	Virtual Machine

WAL	Write-Ahead Log

Table of Contents

1. Architecture

1.1. Introduction

DAIAD comprises several software components and APIs that use data from several sources including water consumption data, user profiles, and system settings, to perform data analysis and visualization. Data from external sources (*e.g., weather, demographics*) is also used to augment the data analysis tasks. DAIAD's Big Water Management Engine is the central component of the DAIAD architecture that stores, manages and provides all this data to *all DAIAD components and APIs*. Its implementation allows supporting a diverse set of query workload types ranging from low latency real-time queries, to high latency offline analytical queries. Moreover, it implements ETL and cleaning processes for integrating data from either external data sources or the output of data analysis jobs.

1.2. Requirements

Data management is one of the most important aspects of the DAIAD system, guaranteeing optimal performance, high availability and low latency query responses. In Deliverable D1.2 "DAIAD Requirements and Architecture" we identified two distinct query workload types that need to be supported from our Big Data engine: (a) *real-time low latency* queries, and (b) *high-latency analytical* queries. Specifically, the following basic query workloads have been identified:

- A continuous stream of write only queries with sensor data from DAIAD@home. As long as no extensive read queries are submitted simultaneously, key value or wide column NoSQL data stores tend to handle such workloads efficiently.
- Numerous read only queries that request a small portion of the stored data. For example, a user requests to download historical consumption data for a specific interval, browses analysis results and recommendations about her own consumption behavior.
- Numerous read/write queries that update user preferences and settings, configure notifications, initialize analysis jobs and retrieve administration information.
- Read queries that involve scanning large portions of the stored data at a high granularity level. Such queries are usually part of the execution of complex analysis algorithms and Map Reduce jobs.

For the first three workload types, responses are expected to have *low latency*. Otherwise, user experience may decrease significantly. The last workload is expected to have high latency but this is acceptable since it involves querying vast amounts of data. Nevertheless, if interactive analysis is expected (e.g. implementation of the what-if analysis module), hierarchical pre-aggregation and data filtering may be further applied to limit the size of input data.

Running both low and high latency workloads against a single data store may result in unpredictable performance behavior, especially for the queries that require low latency responses. For instance, load spikes are expected to be highly probable for queries originating from DAIAD@home. Starting a resource expensive MapReduce job simultaneously may force DAIAD@home clients to resubmit failed uploading requests. A scenario like this is quite likely to occur and opposes to DAIAD's principle of collecting measurements data timely with the minimum impact on user devices. Moreover, adding more servers to the cluster will scale the cluster's performance uniformly for all workload types. Thus, it will be hard to quantify the benefits for each workload type separately. In contrast, it is preferable to scale specific workload processing, e.g., add additional servers for handling data analysis only.

To enhance performance for both query types, we had suggested the implementation of workload isolation. To that end, we deployed and configured two separate clusters, one for handling continuous streams of measurement data, and one for handling the rest of the workload types. An instance of the DAIAD Big Water Data Management engine is installed on each cluster and data is exchanged between the two instances using both online replication and batch data processing jobs. The suggested configuration increases redundancy and disk space requirements. To alleviate this problem, analysis performance can be traded for disk space by lowering the replication factor of the second store. The performance tradeoff is due to the fact that the fewer data replicas are, the less parallelization is attained. Data migration from one data store to the other is performed using batch jobs executing in regular intervals or dynamically by using appropriate cluster topology and configuration.

At the first stages of the DAIAD prototype development, all data had been stored in the HBase NoSQL database, including user account and profile data. As development progressed, many new domain entities have been created, such as data analysis generated clusters, custom groups of users, members of groups, user specific alerts, recommendations, etc. These types of data tend to have many *inter-dependencies* and *frequent updates*. Moreover, they are queried using complex expressions involving join operations and need to be updated *atomically* using transactions. Using HBase for these tasks increased the required development effort and yielded negligible performance gains. To that end, we included the PostgreSQL relational database as a part of our engine to manage these types of data. Whenever a user interacts with a DAIAD application, PostgreSQL is responsible for all requests that do not require water consumption data such as authenticating a user, or accessing a user's profile. It also stores information for controlling *access* to water consumption data stored in HBase by maintaining user application roles. Similarly, most data analysis job are driven by PostgreSQL data, with data exported to files and stored in a distributed file system for the actual processing to take place.

Although using a *mixed persistence model* increases redundancy and deployment complexity, it also achieves *query workload isolation* and assigns data to the *most appropriate data store* depending on its type and usage.

1.2.1. 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.

1.2.2. Real-time Queries

In order to categorize common data access patterns, we enumerate a few common queries expected by the DAIAD 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.

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, the user's 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 "cf".

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.

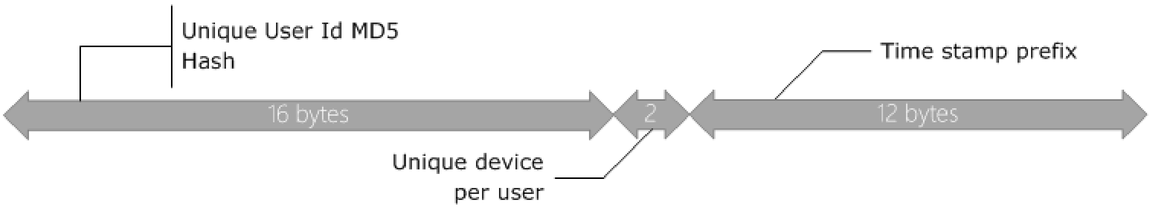


Figure 1: Initial row key schema for amphiro b1 shower data

The latter approach is similar to the technique used by OpenTSDB 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 1 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.

Finally, using the hybrid schema, measurements are 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 2, we decided to create a single column qualifier per measurement in order to accommodate for different metrics or devices i.e. smart water meters.

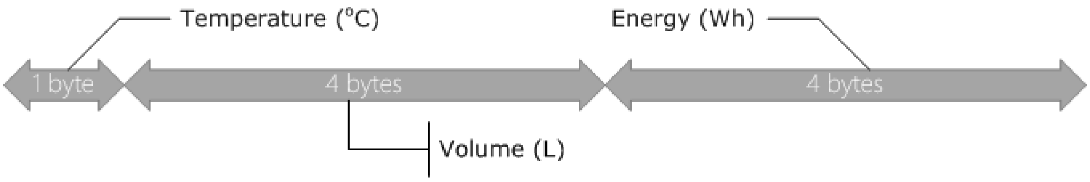


Figure 2: Byte array of a single amphiro b1 shower

In Figure 3, a single cell with two different measurements at timestamps t1 and t2 is depicted.

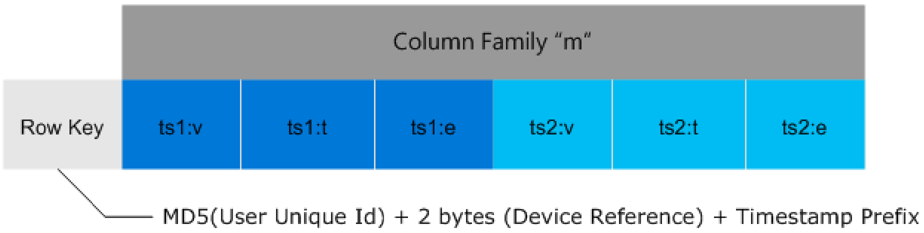

Column Family "m"
Row Key
ts1:v
ts1:t
ts1:e
ts2:v
ts2:t
ts2:e
MD5(User Unique Id) + 2 bytes (Device Reference) + Timestamp Prefix

Figure 3: HBase row with two measurements using composite column qualifiers

1.2.3. Analysis Queries

In the previous section we presented schema design solutions for queries expected by the DAIAD@home applications. In this section, we examine several data schemas focused on data analysis provided by the DAIAD@utility application. In general, user applications tend to submit user-oriented queries. On the contrary, in analysis applications users 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 (*e.g., user age, education*). 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 a schema similar to the one 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 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. 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 using z-ordering or GeoHash[1] for prefixing the row key. In any other case, redundant storage is required. Finally, for ad-hoc analysis queries for which users expect low latency, we employ pre-aggregation on different levels over the time dimension. Hence, initial data with time granularity in seconds is aggregated per user and device over hourly, daily and monthly intervals.

1.3. Architecture

In Figure 4 we depict the stack of software components of the DAIAD Big Water Data Management Engine. Its major components are:

- **Hadoop Distributed File System (HDFS)** provides reliable and redundant storage for all components located higher in the software stack, and is the central piece of DAIAD’s data management scheme. HDFS is used for storing HBase tables, intermediate analysis results and exported data. It also acts as an accessible active archive for storing legacy and backup data.
- **HBase** is a NoSQL database that offers low latency, random data access over HDFS. HBase stores all measurement data generated by sensor devices (amphiro b1) and smart water meters. Moreover, HBase stores and indexes data generated by the analytical processing of measurement data. A presentation of HBase tables and their schema is available in Annex I: HBase row schema.
- **PostgreSQL** is a relational database used for storing frequently changing data, like community memberships, user profiles and preferences, application metadata, cluster management information such as job execution metadata, etc. In general, PostgreSQL is used for storing tables that contain only a few hundreds of thousands of records. The goal of using a different store for this type of data is to minimize the delete operations on HBase. Moreover, PostGIS, a spatial extension of PostgreSQL, is used for managing spatial information associated with users and devices. A full description of the database schema is provided in Annex II: PostgreSQL table schema.

---

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

- The **YARN** resource manager coordinates the execution of jobs from the Hadoop MapReduce and Flink data processing frameworks. Both frameworks can use any of the available storage platforms, namely, HDFS, HBase and PostgreSQL, as a data source and as a data sink.

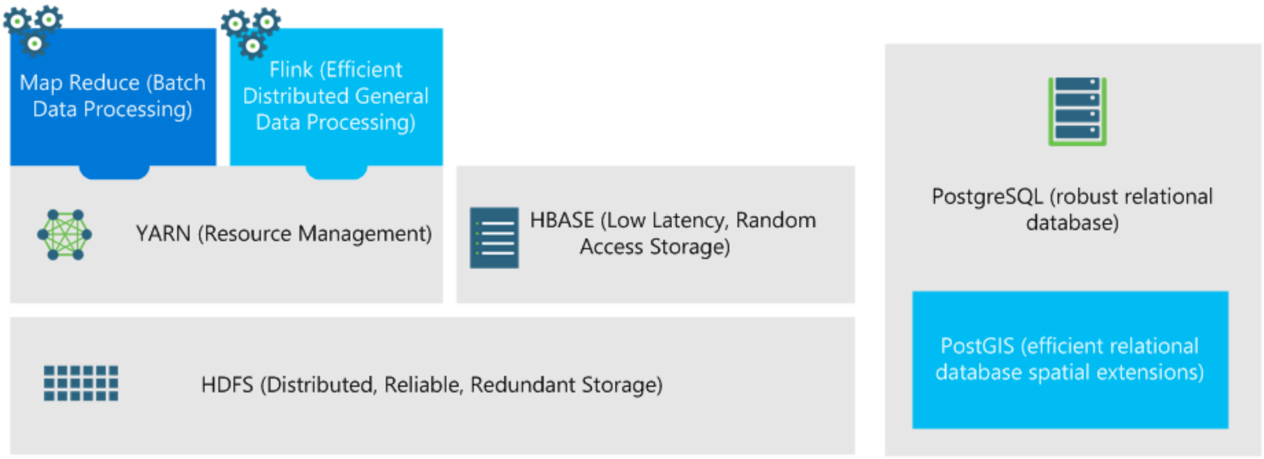

Map Reduce (Batch Data Processing)
Flink (Efficient Distributed General Data Processing)
YARN (Resource Management)
HBASE (Low Latency, Random Access Storage)
PostgreSQL (robust relational database)
PostGIS (efficient relational database spatial extensions)
HDFS (Distributed, Reliable, Redundant Storage)

Figure 4: DAIAD Big Water Data Mananagement Engine Architecture

The Hadoop MapReduce and Flink data processing frameworks are built at the top of the software stack. Both frameworks utilize YARN for controlling resource management[2] and can use HDFS and HBase both as a data source and a data sink. Hadoop MapReduce is optimized for batch processing of large 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 latter feature makes Flink a candidate for completely removing Hadoop MapReduce from our software stack Nevertheless, we have decided to use Hadoop MapReduce for implementing batch jobs such as data importing and aggregation due to the strong integration between HDFS, HBase and Hadoop MapReduce.

Finally, alongside the rest of components is the PostgreSQL relational database. PostgreSQL is used for storing low cardinality, frequently changing datasets like user accounts, smart water meter registrations, user profiles and system settings. By “small” we refer to datasets whose size does not exceeds a couple of millions of records. Moreover, the PostGIS spatial extensions for PostgreSQL are installed for handling spatial data such as smart water meter locations and municipal administrative boundaries.

1.4. Engine components

As pointed out in Deliverable 1.1 “State of the art report”, NoSQL databases are not always replacing existing database or data warehouse solutions, but instead complement them, resulting in mixed persistence models. Moreover, depending on the anticipated query workloads, a solution may be more appropriate than another. In the previews section we presented the architecture of our Big Data Engine with its diverse persistence model using both NoSQL and relational database solutions. In this section we first present the data

---

[2] Hadoop MapReduce and Flink can operate independently from YARN but the latter offers better cluster resource control and utilization.

persistence models applied in our Big Data Engine, and the detail the data engines and related processing frameworks we have applied.

1.4.1. NoSQL

NoSQL (also named *Not-Only-SQL*) databases attempt to solve the problems on storage and processing of Big Data compared to traditional RDBMs. To achieve that, they are designed with three principles in mind, namely, (a) **simplicity** and schema flexibility, (b) horizontal **scalability** and (c) high **availability**.

Starting with simplicity and schema flexibility, NoSQL databases allow to store different data types, including *unstructured* or *semi-structured* data (e.g. text files, geospatial data, web user click-streams, social media data), and to modify data schema dynamically. Hence development of applications that use such data is greatly simplified. As mentioned in the previous section, an RDBMS would require extensive schema updates in order to accommodate the new data types which may also lead to significant system downtime. In contrast, new data types can be added to a NoSQL database without disrupting existing applications, resulting in increased application development agility. Moving to scalability, a system can be either *scaled up* or *scaled out*. In the former case, scalability is achieved by using more powerful and expensive hardware. In the latter, the same result is reached by using cheaper commodity servers. NoSQL databases are designed to be *distributed* and easily scaled out. Using a set of standard, physical or virtual servers for storing and servicing data, scaling out is as easy as adding new servers to the cluster. Therefore, NoSQL database installations tend to more *cost-effective* than RDBMS implementations which offer similar performance characteristics. Finally, most NoSQL database implementations offer data replication, caching and load balancing out of the box hence allowing for high availability.

NoSQL databases can be classified based on different properties such as performance, scalability, data model etc. The most common classification is based on the underlying supported *data model* resulting in four main categories enumerated below. An extensive description of each category is presented next.

- **Key-Value Stores**. Data are stored as key value pairs in an associate array. This is the simplest data model available.
- **Wide-Column Stores**. Store data in records with a dynamic number of columns resembling multi-dimensional key-value stores.
- **Document Databases**. Store documents of a specific format (e.g. XML, JSON) identified by a unique key. Documents can be queried either by their key or by their contents.
- **Graph Stores**. Store data that can be represented as graphs where elements have an undetermined number of relations between them.

The difference in the data model allows for some operations to be faster on a NoSQL database than on an RDBMS depending on the processing task under consideration. For example, in a process that requires the use of graph algorithms, a graph NoSQL database may be more appropriate. In addition, most NoSQL implementations *do not fully adhere to the ACID properties*. Relaxing ACID requirements leads to increased performance but removes support for transactions. Moreover, in contrast to using the declarative SQL language, most NoSQL implementations are programmable through *object-oriented* Application Programming Interfaces (APIs) in various languages (e.g. C/C++, Java, C#) or offer RESTful APIs. Hence, the decision of selecting a NoSQL implementation must be taken after careful consideration of the characteristics of the data persistence problem at hand.

Another important aspect of NoSQL databases is the consistency model they employ. In distributed systems, like NoSQL databases, data is partitioned and replicated across many servers in order to achieve availability and fault tolerance. The consistency model dictates how the system reaches a consistent state (i.e. all reads operations get the most current data) after an operation. RDBMS opt for a strict consistency model where every read operation returns the most recently written value. In contrast, most NoSQL databases compromise consistency in favor of high availability and fault tolerance, and implement a relaxed consistency model, named *eventual consistency*. The eventual consistency model guarantees that, if no updates are executed on a particular data item, all replicas will become synchronized and *eventually* all read operations on that item will return the most recently update value.

Before discussing the different types of NoSQL databases, it is important to notice that NoSQL databases can act complementary to RDBMS technology, instead of always trying to replace it. In the past years, developers used to pick a single persistence model and use it throughout the application lifecycle. With the introduction of NoSQL databases, an application may be using more than a single storage model, based on the data usage scenarios and processing requirements. Mixing different storage models will always come with an extra implementation cost, but with tangible cost-efficiency and scalability benefits.

1.4.1.1. Key-value stores

Key value stores are the simplest form of NoSQL databases. Data is stored in key-value pairs in an associative array with each key being unique. Every implementation defines the format of key values but data values can be anything like text, images, JSON/XML/HTML documents, backups, log files, and more. Different variations exist based on key ordering and type of storage used. For instance, ordered key value stores maintain keys in lexicographic order, hence allowing efficient processing of key range queries. Other implementations favor maintaining data in memory and postpone data persistence on disk when data loss is acceptable by the usage scenarios.

In what follows, a few example implementations of key value stores are enumerated. The list is not complete by any means but illustrates some of the features most implementations offer.

- **Redis**[3]. An advanced key value store that supports multiple key types such as strings, hashes, lists, sets and sorted lists. Redis delivers outstanding performance by implementing an in-memory dataset. Users can opt for disk persistence by periodically dumping memory dataset to disk, or logging all write operations in order to replay them on server start-up. Among other features, atomicity and eventual consistency are supported.
- **Memcached**[4]. A very simple and generic in-memory key value store mainly focused on speeding up web applications by caching data from database calls, external API calls, rendered pages, etc. Memcached supports neither replication nor persistence. Still, performance is maximized and all operations have constant complexity, with every command taking roughly the same time to process every time.

---

[3] http://redis.io/

[4] http://www.memcached.org/

- **Riak**[5]. A NoSQL implementation that offers high availability, scalability and fault tolerance by employing eventual consistency. Value data are versioned and every time the most recent version is returned.

1.4.1.2. Wide-column stores

Wide column stores store data in records with a dynamic number of columns resembling multi-dimensional key-value stores. Alike to document stores, they are schema free; still their implementation is quite different.

Google's Bigtable [CDG+06] is considered to be the predecessor of most wide-column store implementations. Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size of data across thousands of commodity servers. Stored data is indexed by row key, column key and a timestamp that allows multiple versions of the same data. Moreover, data is lexicographically ordered, implementing a persistent multi-dimensional sorted map. The latter allows clients to control the locality of their data through careful choices in their schemas, which accelerates the processing of range queries by decreasing the number of disk accesses and server communication.

A short description of the most popular wide column stores is presented next.

- **Cassandra**[6]. Cassandra is a massively scalable NoSQL database for managing large amounts of structured, semi-structured, and unstructured data and delivering continuous availability, linear scalability, and operational simplicity across many commodity servers with no single point of failure. Data is distributed transparently across all nodes that participate in a database cluster (or "ring" as called in Cassandra terminology). Moreover, if required, Cassandra provides built-in and customizable replication, which stores redundant copies of data across nodes that participate in a Cassandra ring.
- **HBase**[7]**.** HBase is a distributed, scalable, versioned, non-relational data store able to host very large tables sizing to billions of rows with millions of columns, hosted atop a cluster of commodity hardware. HBase is modelled after Google's Bigtable. Just as Bigtable leverages the distributed data storage provided by the Google File System, HBase provides Bigtable-like capabilities on top of the Hadoop Distributed File System (HDFS), a distributed file system designed to run on commodity hardware with linear and modular scalability.
- **Apache Accumulo**[8]. Accumulo extends Google's Bigtable design and is built on top of several Apache projects, namely, (a) Hadoop, a framework that allows implementation of distributed processing of large data sets across clusters of computers using simple programming models, (b) Zookeeper, a service that enables highly reliable distributed coordination, and (c) Thrift, a framework for scalable cross-language services development. Accumulo features a few novel improvements including cell-based access control and a server-side programming mechanism that can modify key-value pairs at various points in the data management process.

---

[5] http://basho.com/riak/

[6] http://cassandra.apache.org/

[7] http://hbase.apache.org/

[8] http://accumulo.apache.org/

1.4.2. Map-Reduce

Processing or generating big datasets is usually a computationally expensive job that takes a long time to complete. Most of the time, such jobs are executed using *parallelism* and *distributed computing* in order to achieve timely responses. Implementing such solutions requires significant effort for splitting the job in manageable tasks that can be processed efficiently by a single server, communicating data between servers, monitoring remote processes, handling software errors or hardware failures, composing the final result from partial responses, etc. It is not unusual that most effort is spent in implementing parallelism and coordinating the execution process than implementing the algorithm that solves the problem under consideration.

MapReduce [DG04] is a programming paradigm and implementation framework that accelerates the development of such solutions by abstracting the execution process management, thus allowing developers to concentrate on implementing an algorithm for their problem. Therefore, programmers can develop efficient, highly distributed solutions without prior knowledge of parallel and distributed systems.

The programming paradigm of MapReduce requires that *at least two methods* are provided for solving a problem, namely, a **Map** method that processes a key-value pair and generates a set of intermediate key-value pairs (usually in a different data domain), and a **Reduce** method that is applied on all intermediate pairs with the same key and returns zero, one or more values.

The associated implementation framework requires the specification of some initialization parameters and thereafter takes care of all execution details including partitioning input data, scheduling tasks to a set of servers, performing load balancing, monitoring process execution, and composing the final result. Moreover, in order to increase parallelism and fault tolerance, a distributed storage system is used and data locality is exploited during data partitioning and scheduling.

A simple word-count MapReduce program is presented below using pseudo code. The program parses a set of text documents stored on several servers and computes the number of occurrences of each word:

```
// key: name of a particular file
// doc: contents of this file
map(string key, document doc) {
        // iterate over all words in document
        for each word w in doc {
                return (w, 1);
        }
}
// key: unique word
// values: iterator that enumerates all intermediate key value pairs (key, 1)
reduce(string key, iterator values) {
        int result = 0;
        // Get next value from the iterator until the list of values is exhausted
        while( values.next()) {
                // Aggregate all values. The value of each key value pair generated by
// the map method is equal to 1
                result += ParseInteger(values.current());
        }
        return result;
}
```

The map function accepts key-value pairs that represent filenames and their corresponding contents and returns a key-value pair for each word it parses with the default value of 1. After the framework orders the intermediate keys, it sums all occurrences for each unique word by calling the Reduce method.

The simple example above describes the work required by a developer. Next, we examine the implementation details of the MapReduce framework.

During the execution of a MapReduce program, the data input is partitioned in M sets. Each set is processed in parallel by different servers, generating intermediate key-value pairs by calling the Map method on each element of the input set. After computation of the Map method, the intermediate results are partitioned into R sets by partitioning the intermediate key using some partitioning function (e.g. hash (k2) modulo R). Finally, the R sets of intermediate results are processed in parallel where the Reduce method is applied. Both parameters M and R are specified by the user. The system has to execute M Map tasks and R Reduce tasks resulting in O (M+R) scheduling decisions overall.

The execution steps of a MapReduce program can be summarized as follows:

- The system splits the input data into M sets. The size of each set can be controlled by the developer during the program initialization. Typically, the set size varies from 16 to 64 megabytes. Selecting a small set size results in manageable input size for a single server and at the same time increases parallelism by exploiting locality of the underlying distributed storage system.
- The framework initializes multiple copies of the program on a set of servers. Those servers are referred as *workers*. One of them is responsible for scheduling the Map and Reduce tasks and is called the *master*.
- Workers that are assigned Map tasks read a set of input data and apply the Map method to each key-value pair in it. The intermediate results are buffered in memory and periodically stored locally after they have been partitioned in R files (or sections) using the partitioning function. Runtime information about the partition location and execution progress is transmitted to the master. In total, M tasks must be processed and M*R sections are generated.
- Workers that are assigned Reduce tasks are informed by the master about the location of intermediate results. After all M_i intermediate results for a specific index in range [1, R] are fetched, they are sorted by the intermediate key and the Reduce function is called for each unique key and the corresponding set of intermediate values associated with it. The result is stored to a final output file in the distributed storage system. At the end of reduce execution, R results files are generated.

Another important aspect of the MapReduce framework is its *automatic fault tolerance*. On the process described above, if a Map worker fails, the task assigned to it is rescheduled, even if it has completed successfully, since the intermediate results are stored locally. On the contrary, a Reduce worker is rescheduled only if not already completed, since the final result is always written on the distributed storage system. In the case the master worker fails, the program aborts resulting in a single point of failure architecture. An alternative solution is to have the master write periodic checkpoints with all the execution status information about scheduling and intermediate result locations. In case of a failure, a new master is restarted at the latest checkpoint.

Finally, additional extensibility points are offered, allowing developers to customize the framework behaviour at runtime. Such features include:

- **Input**: Implement custom readers for reading input data e.g. accessing records in a relational database or key-value pairs from a NoSQL database.
- **Partition**: In general, the default partitioning function generates well balanced partitions. However, there are scenarios that users may require to group the results differently. For example, if the output results are key-value pairs where the key is a URL, it may be preferred that URLs from the same domain are grouped in the same output file. For this case, a hashing function like hash(domain(URL)) may be preferable
- **Combine**: As shown in the pseudo code above, sometimes the intermediate results may contain many values with the same intermediate key. For instance, the pairs ("the", 1), ("a", 1), ("an", 1) may occur thousands of times in comparison to less frequently used words. Transferring these values from the local disks of the Map workers to a Reduce worker and sorting them before running the Reduce method, causes both additional network traffic and computational cost. To remedy this problem, a Combine function can be provided that does partial merging of the data locally at the Map worker before it is sent over the network.
- **Output**: Similar to the custom readers, developers can provide custom writers for writing output files to different formats in order to be consumed by other applications.

1.4.3. Hadoop

Apache Hadoop[9] is a framework for distributed processing of large datasets across a cluster of servers using a simple programming model. Instead of relying on expensive hardware for achieving high availability and scalability, the framework handles hardware and software errors at the application level, thus delivering highly available and scalable computing over a cluster of commodity servers. Hadoop consists of four core modules:

- a set of common utilities that support the other Hadoop modules,
- a distributed file system (HDFS),
- YARN [VM+13], a framework for task scheduling and resource management, and
- a MapReduce implementation based on YARN for parallel and distributed processing of large datasets.

Next we present HDFS, discuss the Hadoop MapReduce implementation and enumerate the main components of YARN.

---

[9] http://hadoop.apache.org/

When data files are loaded in a Hadoop cluster, the Hadoop Distributed File System (HDFS) splits them in smaller blocks of a fixed size and saves multiple copies of each block across several servers in order to guarantee data availability in case of hardware failures. The servers which store each block are chosen randomly on a block-by-block basis. These individual servers are referred to as **DataNodes**. Even though file blocks are replicated and distributed across several servers, the process of partitioning the files and distributing the blocks is transparent to the end users. In order to achieve that, HDFS stores metadata for each file and all of its blocks. In addition, a monitoring service replicates data as needed when system failures occur in order to sustain the system scalability. Hence, if a small percent of servers, e.g. a server rack, goes offline the system performance is affected only by a proportional factor.

The default block size in HDFS is 64 megabytes. Since HDFS expects to store very large files, a large block size decreases the overall size of metadata stored. Moreover, since MapReduce programs usually scan large input files, keeping large amounts of sequentially stored data on a disk results in fast reads. Alternatively, increasing the block size results in less parallelization of Map tasks in a MapReduce job.

The metadata for file partitioning and replication is handled by a single server, namely, the **NameNode**. When a client requests a file, it first contacts the NameNode to access the block locations for the specific file and then retrieves data from the corresponding DataNodes. As metadata size is relatively low (only information about filenames, block locations per file and permissions are kept), the NameNode stores this information in the main memory and can serve a high number of clients with a minimum overhead.

Since metadata is stored on a single server, it is important that it is stored reliably. A NameNode failure is more severe than that of a DataNode since it renders the whole file system inaccessible. In order to scale naming service horizontally, it is possible to declare multiple NameNodes with their corresponding namespaces (file systems). Each NameNode has access to all available DataNodes. NameNodes are independent and don't require coordination with each other. Each NameNode is a single point of failure for the associated file system only. Therefore, the overall availability of the system is increased.

HDFS is used for implementing MapReduce in Hadoop, in the same manner BigTable [CDG+06] is used for implementing Google's MapReduce. The implementation is fairly similar and most of the features described in the MapReduce section apply to the Hadoop MapReduce implementation. The Hadoop framework schedules map and reduce tasks using data locality knowledge from HDFS. Thus computation in moved to the data, instead of moving the data to the computation which results in high performance and less network traffic.

Moreover, Hadoop provides a mechanism for managing batches of jobs with dependencies among them. Instead of submitting a single job, users can declare multiple jobs with dependencies and submit them as a single unit of work. A job won't start before all its dependencies have completed successfully.

Another important aspect of Hadoop MapReduce is that job scheduling and resource management is separated from the MapReduce programming model implementation and handled by another framework, namely, YARN (Yet Another Resource Negotiator). YARN is agnostic to the applications that request resources and allows Hadoop to be used for implementing other computational models apart from MapReduce such as graph processing, machine learning, etc.

YARN uses a single **ResourceManager** for managing collectively the resources of the whole cluster and one **NodeManager** per server for managing the server's local resources. The resources on each server are organized in leases (containers) that represent CPU, memory, disk space, network bandwidth etc. The ResourceManager has two main components, namely, the **Scheduler** and the **ApplicationsManager**.

The ApplicationsManager is responsible for accepting job requests, initializing job execution and monitoring their status in case a restart is required when a failure occurs. The Scheduler is responsible for allocating resource containers to running applications (jobs) and enforcing constraints on the resources utilization. The policy for sharing resources among various applications is pluggable and extensible. An example of such a pluggable scheduler is the FairScheduler that assigns resources to applications such that all applications get, on average, an equal share of resources over time.

The ResourceManager is ignorant of the semantics of each resource allocation requested by an application. For each application (job) running on the cluster an ApplicationMaster is assigned. **ApplicationMaster** is actually a framework specific library that implements a particular programming model e.g. MapReduce and is responsible of negotiating appropriate resource containers from the Scheduler, generating a physical execution plan, monitoring and tracking execution progress and handling execution errors. Before starting a new job, a container must be acquired from the ResourceManager for launching the ApplicationMaster itself. Afterwards, the ApplicationMaster negotiates resources from the ResourceManager and once sufficient resources are acquired, it works with the one or more instances of NodeManager for executing its jobs.

1.4.3.1. Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) is a distributed, highly available and scalable file system, designed to run on commodity hardware and is an integral component of the Hadoop ecosystem. HDFS splits files in blocks which are replicated across a set of servers. The storage servers are called DataNodes. A single server in the cluster, namely the NameNode, is responsible for managing the file system namespace (directory structure), coordinating file replication and maintaining metadata about the replicated blocks. Every time a modification is made e.g. a file or directory is created or updated, the NameNode creates log entry and updates metadata. Clients contact the NameNode for file metadata and perform I/O operations directly on the DataNodes. A high level overview of the HDFS architecture is depicted in Figure 5.

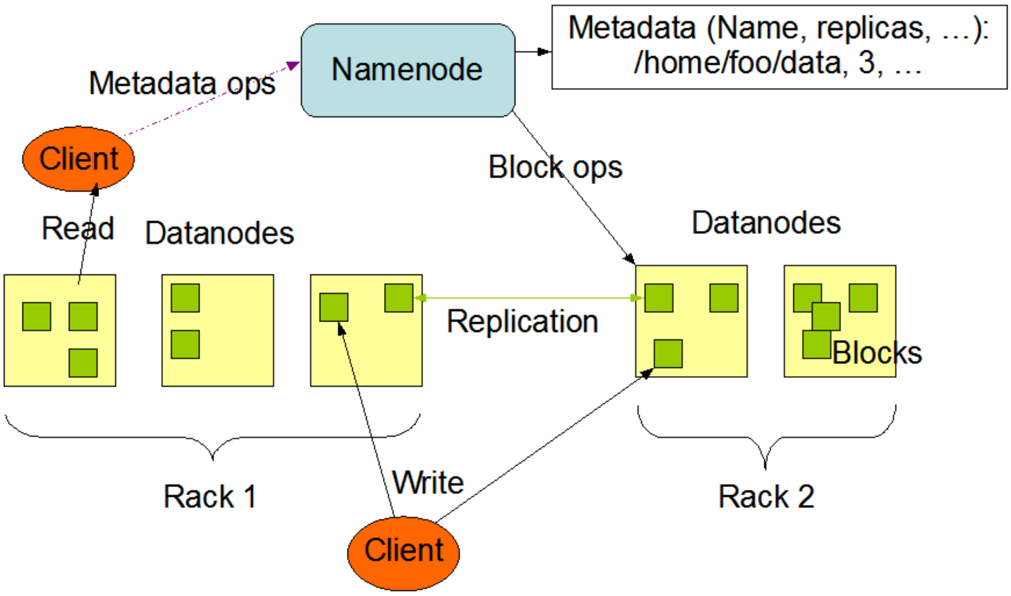


Figure 5: HDFS architecture

The NameNode is a single point of failure in a HDFS cluster. In order to increase availability, the NameNode maintains multiple copies of the metadata and log files. Moreover, an optional secondary NameNode can be deployed for creating checkpoints for the metadata and log files. Creating checkpoints allows for faster recovery times. In addition, HDFS can be configured to use multiple independent NameNodes, thus implementing many autonomous file system namespaces. The latter feature increases I/O operation throughput and offers isolation between different applications.

Finally, HDFS is optimized for managing very large files, delivering a high throughput of data using a write-once, read-many-times pattern. Hence, it is inefficient for handling random reads over numerous small files or for applications that require low latency.

1.4.3.2. Hadoop Map Reduce

Hadoop Map Reduce is a big data processing framework that simplifies the development of highly parallelized and distributed programs. Developing distributed programs requires handling many tasks including data replication, data transfer between servers, fault recovery, management of many parallel executing tasks etc. Hadoop abstracts the complexity of developing parallel and distributed applications by making all the aforementioned tasks transparent, allowing developers to focus on the problem under consideration.

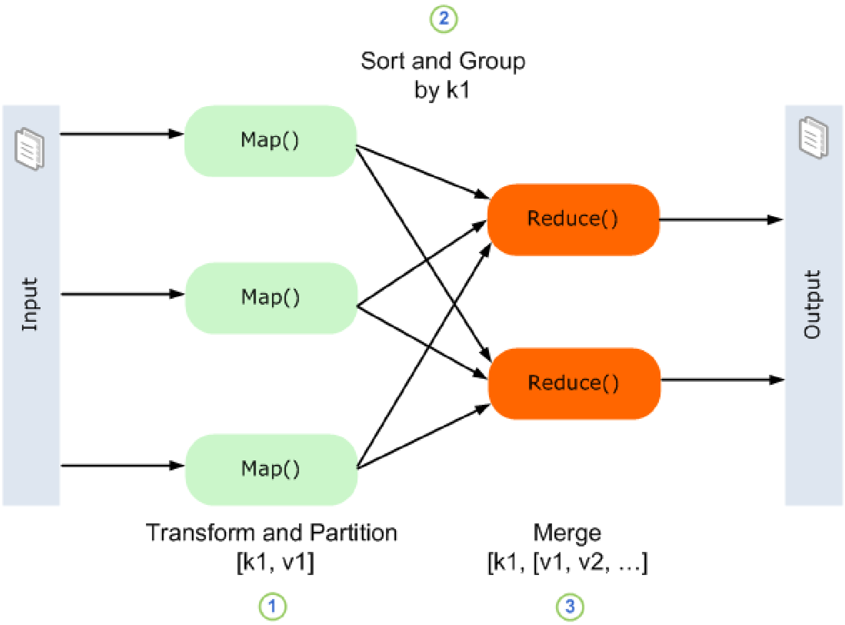


Figure 6: MapReduce computing model

The Hadoop Map Reduce initial version shared an architecture similar to HDFS. In particular, a TaskTracker resided on every DataNode that was responsible for performing computations on the specific server. Similar to the NameNode, a JobTracker was acting as a master node that performed resource management and job scheduling and monitoring. In newer versions of Hadoop Map Reduce, resource management and job scheduling has been assigned to a new component, namely YARN[10]. Therefore, the implementation of other distributed computing models, such as graph processing, over a Hadoop cluster is possible and also Hadoop Map Reduce focuses exclusively on data processing.

---

[10] For a more detailed presentation of YARN and Hadoop Map Reduce architecture refer to Deliverable 1.1

The computing model of Hadoop Map Reduce is depicted in Figure 6. A Map Reduce program requires the implementation of two methods, namely Map and Reduce. The Map method transforms input data to a set of intermediate key value pairs which are partitioned on the generated key. Then, the intermediate partitioned key value pairs are sorted and grouped by the generated keys. Finally, the created groups are processed by the Reduce method and the final output is produced. Both intermediate and final results are stored on HDFS.

1.4.4. HBase

Apache HBase is a free, open source, distributed and scalable NoSQL database that can handle tables with billions of rows consisting of millions of columns. HBase is built on top of HDFS and enhances it with real time, random read/write access.

The architecture of HBase is similar to that of HDFS. Table data is organized in regions that are managed by a set of servers, namely RegionServers. Usually a RegionServer is installed on every DataNode of the underlying HDFS storage cluster. By default, each region is served by a single RegionServer. Still, HBase can be configured for region replication if availability is more important than consistency. Fault tolerance is attained by storing HBase files to HDFS. Likewise, an HBase Master node is responsible for monitoring RegionServers and load balancing. Usually, the HBase Master is installed on the same server with HDFS NameNode. In addition, more than one HBase Master may be present in a master/salve configuration in order to circumvent single point of failure issues. Finally, Apache ZooKeeper is used for coordinating and sharing state between master and region servers. Clients connect to ZooKeeper for submitting requests and read and write data directly from and to the region servers.

HBase integrates seamlessly with Hadoop Map Reduce framework since they share the same underlying storage. Moreover, since rows are sorted by row key, HBase scales well for both fast row key scans across tables as well as single row read operations. Hence, HBase can be efficiently used both as a Hadoop Map Reduce source as well as a data store for ad-hoc querying and low latency applications.

1.4.1. Flink

Apache Flink (*formerly Stratosphere*), is a general purpose, distributed and efficient data processing framework that provides an extensive set of operators for expressing complex algorithms. Flink operators are designed to work in memory and gracefully fallback to external memory algorithms once memory resources become scarce.

Instead of building its functionality on top of an existing MapReduce framework, Flink implements its own job execution runtime. Therefore, it can be used either as an *alternative* to Hadoop MapReduce component or as a *standalone processing system*. When used with Hadoop, Flink can access data stored in HDFS and request cluster resources from the YARN resource manager.

Flink extends the MapReduce programming model with additional operators, also called transformations. An operator consists of two components, a user-defined function (UDF) and a parallel operator function. The operator function parallelizes the execution of the user-defined function and applies the UDF on its input data. The data model used by Flink operators is record based instead of the key-value pair model used by MapReduce. Still, key-value pairs can be mapped to records. A collection of records is referred to as dataset. All operators will start working in memory and gracefully fallback to external memory algorithms once memory resources become low. The new operators represent many common data analysis tasks more naturally and efficiently

A short description of the available operators as presented in the Flink documentation follows.

- **Map**: The Map transformation applies a user-defined function on each element of the input dataset. It implements a one-to-one mapping, that is, exactly one element must be returned by the function. This behavior differs from that of the classic Map operator.
- **FlatMap**: The FlatMap transformation applies a user-defined function on each element of the input dataset. This variant of a map function can return arbitrary many result elements (including none) for each input element. The behavior of this operator matches that of the classic Map operator.
- **Filter**: The Filter transformation applies a user-defined function on each element of the input dataset and retains only those elements for which the function returns true.
- **Project**: The Project operator allows the modification of the fields of a tuple. A tuple is an ordered list of fields. Users can shuffle, add or remove fields. Project operator does not need the definition of a user-defined function.
- **Reduce** on grouped dataset: A Reduce operator that, when applied on a grouped dataset, it reduces each group to a single element. For each group of input elements, the user-defined function successively combines pairs of elements into one element until only a single element for each group remains. There are different variations for this operator. For example, users may define an additional function for extracting the key used for grouping from each element.
- **GroupReduce** on grouped dataset: A transformation that, when applied on a grouped dataset, it calls a user-defined function for each group. The difference between GroupReduce and Reduce is that the user defined function gets the whole group at once instead of a pair of elements at a time. The function is invoked with an iterator over all elements of a group and can return an arbitrary number of result elements. This operator resembles the classic Reduce operator.
- **Reduce** on full dataset: Applies a user-defined function to all elements of the dataset. Pairs of elements are subsequently combined into one element until only a single element remains.
- **GroupReduce** on full dataset: Applies a user defined function to a dataset by iterating over all the elements of the dataset. An arbitrary number of result elements is returned.
- **Aggregate** on grouped tuple dataset: Supports min, max and sum aggregation operations. The aggregate transformation can only be applied on a dataset of tuples.
- **Join**: Joins two datasets into one dataset. The elements of both datasets are joined on one or more keys which can be specified either by a user defined function or by field indexes, if elements are tuples.
- **Cross**: The Cross transformation combines two datasets into one dataset by building a Cartesian product. The Cross transformation either calls a user defined function on each pair of elements or applies a projection
- **CoGroup**: The CoGroup transformation jointly processes groups of two datasets. Both datasets are grouped on a defined key and groups of both datasets that share the same key are passed together to a user-defined function which iterates over the elements of both groups. If for a specific key of any

dataset there are no matching elements from the other dataset, an empty group is passed to the user function.

- **Union**: Produces the union of two datasets, which have to be of the same type.

Flink allows to model job processing as *directed acyclic graphs (DAGs)* of operations, which is a more flexible model than MapReduce, in which Map operations are strictly followed by Reduce operations. The combination of various operations allows for data pipelining and in-memory data transfer optimizations, which increase performance by drastically reducing disk access and network traffic. Moreover, Flink supports highly efficient iterative algorithms, which are very important for Data Mining, Machine Learning and Graph processing, since such jobs often require looping over the working data multiple times. Implementing such jobs with MapReduce is quite expensive since data are transferred between iterations by using the distributed storage. In contract, Flink supports iterative algorithms in its core.

Flink offers powerful APIs in Java and Scala. The Flink optimizer compiles the user programs into efficient, parallel data flows which are executed on a cluster or a local server. The optimizer is independent of the actual programming interface and supports cost-based optimization for selecting operator algorithms and data transfer strategies, in-memory pipelining of operators, data storage access reduction and sampling for determining cardinalities.

2. Deployment

2.1. Production environment

DAIAD's Big Data Engine is deployed on top of the Synnefo cloud stack[11], within a number of virtual machines. Synnefo is a complete open source cloud stack written in Python that provides Compute, Network, Image, Volume and Storage services, similar to those offered by AWS[12]. On a hardware level, the production system is hosted on Athena RC's own server infrastructure (*240 CPU cores, 184 GB main memory, 40TB storage*) and in two clusters (named "c1" and "c2") divided into 6 virtual node groups each - with the provision of spinning up more virtual machines into each group if necessary, through Synnefo and Ansible. The existing groups are:

- **Administration node** (admin-cX.dev.daiad.eu): This is a single-node group, and acts as a central point for the administration of the entire cluster. It provides terminal access to the internal network of cluster "cX", performs periodic health-checking, maintenance or rolling-update tasks.
- **Data group** (data-cX-nYY.dev.daiad.eu): This node group contains those services that are considered as performing at the data level (e.g. measurements, aggregated results on measurements). It includes HDFS data nodes, YARN node managers, HBase region servers, and Flink task managers. This group typically holds valuable (as possibly irreplaceable) source data, thus services usually operate on a high replication level (~3).
- **Master group** (master-cX-nYY.dev.daiad.eu): This node group contains those nodes that continuously monitor and coordinate the nodes of the "data" group: HDFS name (master/secondary) nodes, HBase master, ZooKeeper.
- **Manager group** (manager-cX-nYY.dev.daiad.eu): This group contains job-managing nodes: YARN resource manager, MapReduce history server, Flink job manager.
- **Application group** (app-cX-nYY.dev.daiad.eu): This group contains nodes that host the web application servers (Tomcat7) and is the only group that has access to the public IPv4 network, as the only one that hosts user-facing (HTTP) services.
- **Relational database group** (rdb-cX-nYY.dev.daiad.eu): The node group that hosts the relational database service (PostgreSQL, enhanced with PostGIS). As mentioned before, the main role of this service is to hold structured user-related data (e.g. location, account info), and is strongly coupled with the "application" group.

---

[11] https://www.synnefo.org/

[12] https://aws.amazon.com/

2.2. Deployment

In this section we present the physical deployment of our Big Data Engine and the description of the processes executed by each host in it. Figure 7 depicts the type of hosts in the cluster, their cardinality and the processes which are executed on them.

- *5xData Nodes*. Each data node is responsible for storing and processing data. Every data node executes an HDFS DataNode process for handling HDFS data, a 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 provide 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 with PostGIS spatial extensions for storing smaller datasets required for application and cluster administration.

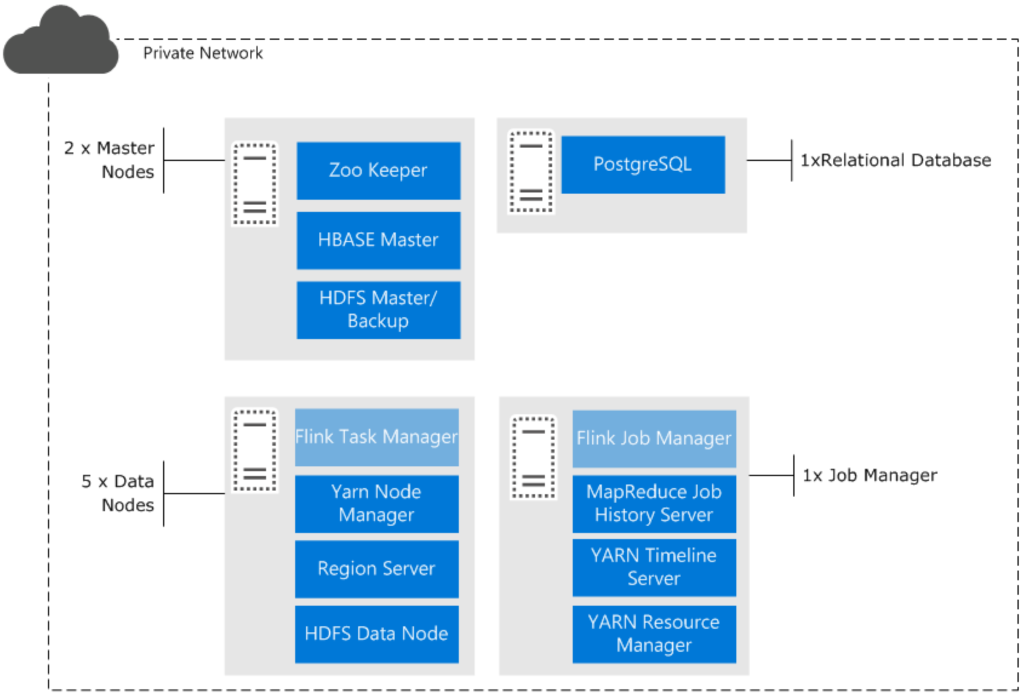


Figure 7: Deployment of DAIAD Big Water Data Management engine (single cluster)

Optionally, an instance of Flink Task Manager per data node and a single instance of Flink Job Manager on the Job Manager node may be installed. Nevertheless, since all jobs are implemented as YARN applications, there is no distinct Flink installation. Moreover, all the software components are installed on every node in the cluster, allowing any node in the cluster to perform any role depending on its configuration. Consequently, we can easily add new nodes to the cluster with minimum configuration effort and initialize only the required services e.g. HDFS data node and HBASE region server for a data node without processing capabilities.

A second cluster is also installed and data replication processes are configured. We refer to the two clusters as real-time data cluster and data analysis cluster. The former is responsible for handling real-time user requests while the latter executes data analysis jobs. The following two methods are applied for data replication:

- For real-time data, we are synchronizing the HBase databases using the Write-Ahead Log (WAL). Updates are propagated from the cluster handling real-time data to the data analysis cluster.
- For data aggregation, the data analysis cluster computes the aggregates using batch jobs and stores the results directly to both clusters.

Although all nodes are equivalent by means of software installed as mentioned earlier, the main differences between the two clusters deployment are the following:

- The real-time data cluster has not dedicated Job Manager node while the Data Nodes have no Yarn Node manager service running.
- The data analysis cluster has no dedicated Relation Database node. Any required data stored from PostgreSQL are exchanged using text files during the data analysis jobs submission.

3. Data Schema

In this section, we present the design of the database schema used for storing data in the Big Water Data Management Engine. First, we briefly enumerate PostgreSQL tables and describe the migration tools used for updating their schema. In the following, the schema for HBase row keys and column qualifiers is presented together with its evolution during the development of the DAIAD system.

3.1. PostgreSQL Schema

PostgreSQL is used for storing low cardinality and frequently changing datasets like user accounts, security roles assignment and smart water meter registrations. A description of the most important tables in the database is provided in Annex 6.

PostgreSQL schema tends to change more often that the HBase one. This happens because the data sources for HBase data are only a few and well known in advance, namely, smart water meters and amphiro b1 devices. Moreover, row key format is changing less often and adding new columns to HBase tables requires updates only to the applications since HBase table columns can be changed dynamically.

To handle the PostgreSQL schema evolution, we applied schema versioning. The versioning is implemented using the Flyway[13] migration tool. The schema migration is performed whenever the application is upgraded and guarantees that the application will fail to start if any schema inconsistencies are detected. This feature protects the data integrity and prevents data corruption due to an outdated database schema. We have also integrated database versioning to the testing of the application. This is an extra layer of security since any error in the database schema will cause the failure of the code build process.

As an extra layer of protection, during the application initialization, the application performs tests to verify that the required data for the current version of the application are in sync with the contents in the database. For instance, if there are no corresponding records in the database for the members of an enumeration, the application fails to start.

Finally, data migration between schema versions is implemented as version schema updates too. Hence, data migration is always performed after a successful schema migration.

3.2. HBase schema

In this section, we present the 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.

---

[13] https://flywaydb.org/

To make well-informed choices we identified the specific data access patterns required by the application use cases. The requested queries have the following characteristics:

- Query requests data for an individual user or a group of users. The size of the group may vary from a small group to the entire city population.
- There is always a time interval constraint of variable granularity e.g. hourly, daily or monthly intervals.
- Depending on the data source the results may contain multiple measurement values i.e. volume and temperature in the case of amphiro b1 data or minimum and maximum consumption per user in the case of smart water meter data.

In addition to the query workload, our solution attempts to achieve the following goals:

- Distribute the data evenly across all region servers to avoid creating performance bottlenecks for read/write operations
- Minimize the index size in comparison to the actual data to preserve storage space.
- Preserve data locality to accelerate range scan queries.
- Try to simplify row cell values access by trading longer row key size for simpler column qualifiers and less number of columns per row.

In the final implementation, we have decided to created two redundant copies of data and store consumption data in two tables. The first table schema is optimized for single user data access and is indexed based on the smarter water meter serial number while the second is designed towards to supporting range scan queries for data pre-aggregation and is indexed based on the measurement timestamps. Next, we refer to these tables as user based indexed[14] and time based indexed respectively.

For the user based indexed table the row key is depicted in Figure 8. The key is composed by the following parts:

- The MD5 hash of the smart water meter unique serial number. Using a hash value instead of the actual value helps to achieve better data distribution across region servers. In addition, hashing guarantees that the row key is of fixed size which simplifies row key management during development.
- The timestamp of the measurement rounded to the day level. Rounding timestamps allows saving and retrieving all the measurements for a whole day interval inside a single row. Moreover, since the users tend to query most recent data more frequently, timestamps are also stored in reverse order.

---

[14] The data is indexed based on a SWM serial number. Nevertheless, since a SWM is always assigned to a single user we refer to this table as user based indexed.

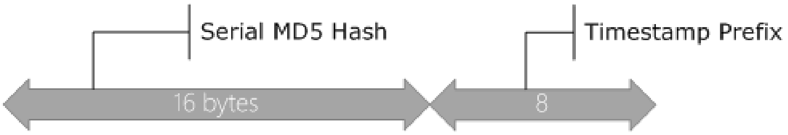


Figure 8: Row key format for the user based index table

For the time based indexed table the row key is depicted in Figure 9 and key is composed by the following parts:

- The remainder of division (modulo operation) of the timestamp by the number of region servers. Using this prefix creates a more balanced row distribution. Otherwise, if the timestamp was used as the row prefix all insert operations will be served by a single region server node which is something we need to avoid.
- The timestamp of the measurement rounded to the day level like the user based index table row key.
- The MD5 hash of the smart water meter unique serial number.

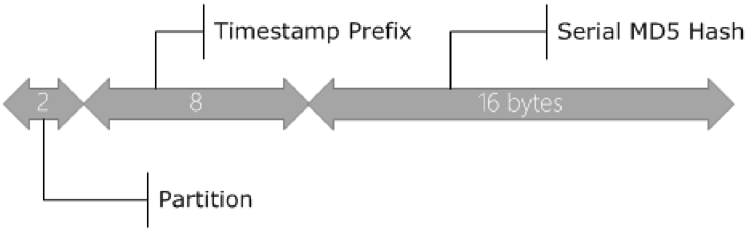
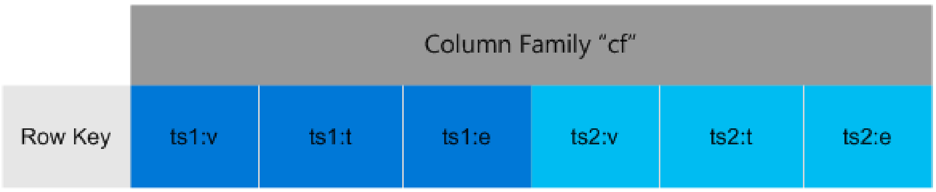


Figure 9: Row key for the time based indexed table

With the format of row keys set, 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 cell value. Using short column family names decreases both storage space and network traffic. Moreover, it is encouraged not using many family columns. In DAIAD, only one column family, namely “cf” is used.

Finally, the column qualifiers are composed by the timestamp offset and the name of the measurement. Using composite column qualifiers, we handle the cell values inside a physical row as a set of logical rows. In Figure 10, a single row with three measurements and two logical rows at timestamps t1 and t2 is depicted.



	Column Family “cf”					
Row Key	ts1:v	ts1:t	ts1:e	ts2:v	ts2:t	ts2:e

Figure 10: HBase row with three measurements and two timestamps

The user based index table allows to query efficiently the data of any individual user. However, answering queries for a group of users requires the execution of as many scan operations as the cardinality of the group. For large groups this is evidently not efficient. Consequently, the user based index table is used mostly for executing queries for DAIAD@home applications. Answering queries over multiple users and variable time intervals is handled by the time based indexed table. Nevertheless, whenever a scan operation is performed against this table, all the row keys for a specific timestamp prefix must be scanned to select the appropriate SWM serial numbers. Hence, the smaller the cardinality of the user group is, the more irrelevant rows are fetched that do not affect the query result. To solve this problem, we employ data pre-aggregation. Data is aggregated using a MapReduce job that is scheduled for daily execution. The time based indexed table is used as input which allows for the incremental update of the aggregated data without the need to scan the entire table. The results of the pre-aggregation process are stored in a third table whose row key schema is depicted in Figure 11.

The row key consists of the following components:

- The aggregation level which can be day, week or month.
- The remainder of division (modulo operation) of the timestamp by the number of region servers.
- The timestamp of the left boundary of the time interval rounded to the aggregation level.
- The MD5 hash of the smart water meter unique serial number or the unique key of a user group. A user group can be either a whole utility, a cluster of users or a DAIAD@commons community.

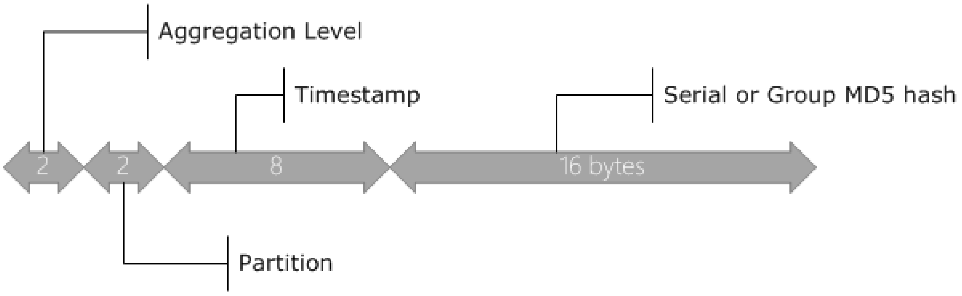


Figure 11: Row key for aggregated data

During the development of the DAIAD project we had to revise the HBase row keys several times. Yet, converting data to the new schema is not done as easily as in a relation database due to the custom format of the row keys. Whenever updates had to be applied to the schema, we created a new table with the name of the old one appended with the version number. Next, the data from the old table were loaded, transformed and imported into the new table using either a Map Reduce job or a custom job.

4. Data Import and APIs

In this section, we present in detail all implemented ETL facilities for importing, cleaning, and exporting DAIAD data, and the Data API, which exposes data management and processing services across all sub-systems and components of the complete DAIAD system. The ETL processes extract data from diverse data sources of numerous formats and integrate it into the Big Water Data Management Engine. During the integration process, data is validated to preserve the data store's integrity and consistency. The Data API provides access to the stored data through a consistent and well defined programmatic interface to both internal DAIAD components and external applications. It enforces application security policies and acts as a single version of the truth for all DAIAD data.

4.1. ETL

We have developed and deployed several ETL processes for the curation and provisioning of data collected from various sources. ETL processes are used for both importing external data, like smart water meter data, and exporting stored data in appropriate formats for further processing and distribution among the members of the DAIAD project and trial participants.

An ETL process may use the Big Water Data Management Engine as a *data source, a target, or both*. For instance, external data such as meteorological data, may be imported into the store, smart water meter data can be exported to files from the store, or data in the store may be just transformed during a HBase schema migration. In every scenario, an ETL process always preserves *data integrity and consistency*, prevents invalid data to reach the store, and always exports data of the highest available quality.

ETL processes provide the DAIAD system with data required by the data analysis components, such as demographic data and water consumption data from water utilities. However, data from 3rd parties may contain errors which may result to error propagation to the results of data analysis facilities. Moreover, the data format may be incompatible with the data store or missing specific data properties and metadata. ETL processes validate data, remove any errors and augment data before importing it. Moreover, ETL processes generate bulk exports of the data in well-defined formats which cannot be done efficiently using the Data API.

A high-level overview of the data sources used by the ETL processes is depicted in Figure 12. All the data is eventually stored at the Big Data Management engine.

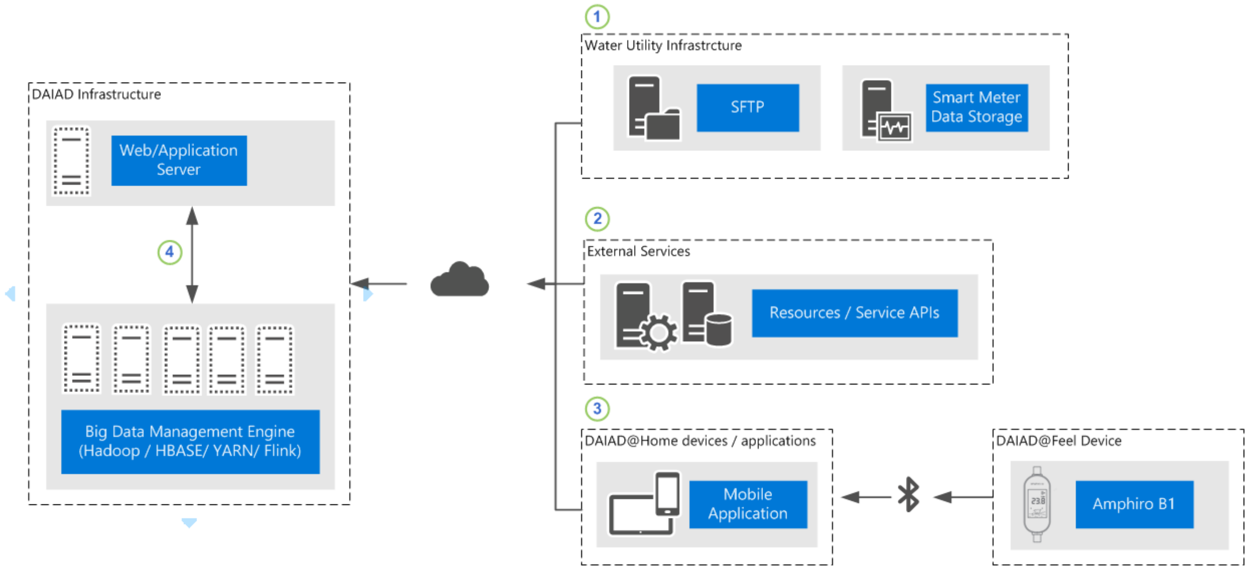


Figure 12: ETL Data sources

Currently, the implemented ETL processes manage data from the data sources enumerated below. The list is not exhaustive since the developed infrastructure is modular and can be extended easily to accommodate new data sources.

- Historical smart water meter data which is downloaded periodically from a SFTP server.
- DAIAD@feel generated data which is uploaded by the DAIAD@home mobile application using the appropriate API endpoint[15].
- Weather data time-series harvested from public or commercial services like AEMET[16] and Weather Underground[17] respectively.
- The results of the execution of data analysis algorithms developed in WP5, "Big Water Data Analysis".

An overview of an ETL process structure and its execution environment is illustrated in Figure 13.

---

[15] https://app.dev.daiad.eu/docs/api/index.html#api-Data-DataStoreByIndex

[16] http://www.aemet.es/en/

[17] https://www.wunderground.com/weather/api/

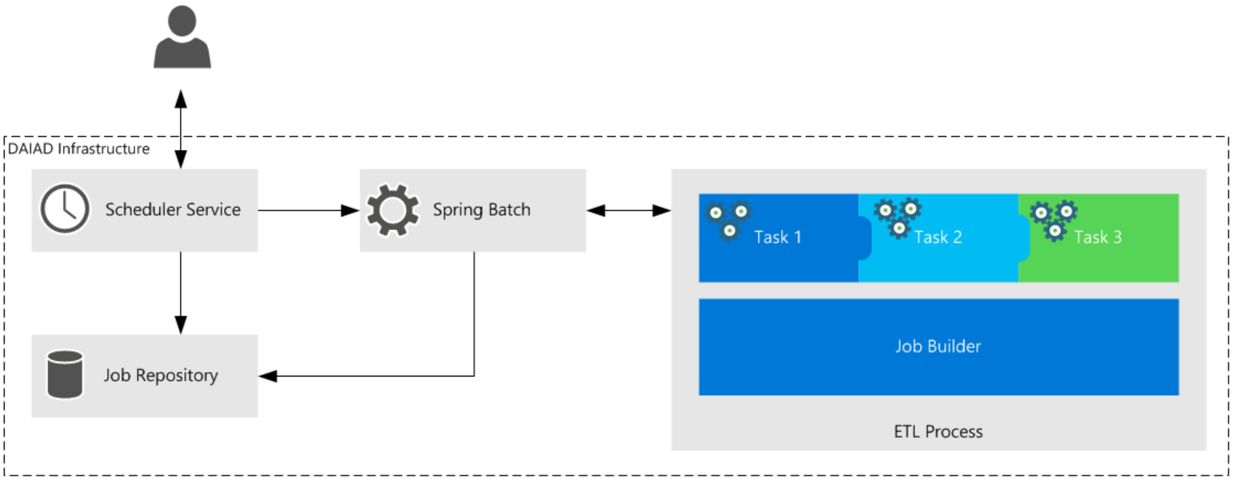


Figure 13 : ETL process structure, scheduling and execution

On the right side of the figure, the structure of an ETL process is shown. Every ETL process in DAIAD consists of one Job Builder component and one or more tasks components.

A task component represents a single unit of work like creating a directory, importing a CSV file with SWM data in HBase or fetching data from a remote SFTP server. A task is completely agnostic of its execution context and the only interacts with it to request job parameters or to publish new parameters. For instance, a simple task that creates a directory requests a root path as a parameter from the execution context and publishes the path of newly created directory if it completes successfully. Tasks are designed with reusability in mind and support extensive parameterization. Thus, a task component can be used as a building block for any ETL process.

The Job Builder component is responsible for configuring and orchestrating the execution of several task components. Usually a new Job Builder component is implemented for every individual ETL process. Yet, multiple ETL processes can be registered using the same Job Builder component if their parameters differ. For instance, an ETL process that downloads SWM data from an SFTP server and imports it in HBase, can be registered and invoked several times simultaneously for fetching data from different SFTP servers.

The creation and execution of Job Builder is handled by the Spring Batch Framework[18]. Moreover, the scheduling and execution of ETL processes is decoupled with a separate scheduler service being responsible for scheduling. The scheduler service can either configure a process to execute at specific time intervals or initialize the execution of a process on demand after a user's request. In either case, the Spring Batch Framework guarantees that no duplicate executions will exist. The latter feature ensures that no inconsistencies are introduced in the data due to concurrent data write operations.

All required Job Builder parameters are stored by the Job Repository component in a PostgreSQL database instance. Both scheduler service and Spring Batch access the Job Repository to discover registered Job Builder components, read job parameters and write job execution status information. Decoupling the Job Repository

---

[18] http://projects.spring.io/spring-batch/

from the scheduler and execution engine allows the two components to reside on separate hosts and execute long running offline jobs.

Finally, the ETL processes are executed offline using the scheduler service. The only exceptions are the import of DAIAD@feel shower data and the export of water consumption data for individual users as explained in the next sections.

4.1.1. User Data Export

The User Data Export ETL process exports water consumption data from all registered SWMs and amphiro b1 appliances for an individual user. The process consists from only one task that (a) retrieves the data from HBase, (b) copies it in an Excel file and (c) compress the final file using the ZIP archive format. The data is retrieved from the user based index table described in section 3.2. Since the size of the data for an individual user is relatively small, the exporting operation is very efficient. To that end, this process does not execute offline. Instead, the Spring Batch framework is bypassed and the corresponding task component is instantiated and invocated immediately.

4.1.2. DAIAD@feel Data Import

Like the User Data Export task, the DAIAD@feel Data Import process is executed without the intervention of the scheduler service and the Spring Batch framework. Instead, whenever new shower data is received the following actions are executed:

- All shower measurement time-series values are associated to a shower in the request. Measurements that do not belong to any shower are ignored.
- For every shower, the HBase row key is computed and any existing row is retrieved.
- Depending on whether the shower has real-time or historical data, the existing row is updated or a new row is created.
- For new showers, all shower measurement time-series values are inserted into HBase. If a row for a shower already exists, the existing time-series values are merged with the new one.

A shower may be received multiple times. Yet, the process always merges the values with existing ones and guarantees that the most accurate data is stored.

4.1.3. Weather Time-Series Import

Weather Time-Series Import process is scheduled for daily execution. It is composed by a single task component that collects weather data from one or more weather services. During the execution, the following actions are performed:

- The PostgreSQL database is queried for any registered weather service providers.
- A list of supported utilities is compiled for every provider. For instance, the AEMET service provides data for the Alicante utility but not for St. Albans one.
- For every utility in the list the corresponding provider is queried for forecasting data for the next seven days.

- The provider's response is merged with any existing data from the same provider. Thus, the accuracy of the data is continuously increasing.

The final data can be queried by DAIAD client using the appropriate weather API[19].

4.1.4. Smart Water Meter Data Import from SFTP

During the trial in Alicante, AMAEM is collecting SWM data for the participants and uploads it to a SFTP server. The Smart Water Meter Data Import process is responsible for downloading, cleaning and importing this data to the HBase to become available for querying and analysis.

The corresponding Job Builder implementation instantiates and executes the following tasks in the given order:

- A random temporary folder is created to isolate the intermediate input files from any other process. This directory serves as the working directory for the rest of the process.
- The AMAEM SFTP server is queried for new files and if any are found, they are downloaded in the working directory.
- The files in the working directory are sorted in chronological order and are parsed sequentially. Every file is processed in chunks of approximately one hundred thousand rows each to avoid excessive memory usage. After a chunk is parsed, all the rows are validated and imported to HBase. Whenever the process ends parsing a file, it saves a copy of it for future reference and stores several statistics in PostgreSQL database like total rows parsed, total rows imported and total rows skipped.
- After the completion of the process the working directory is deleted.

During the import operation, the process validates the row input format and numerical values. For every row four fields are parsed, namely, the SWM serial number, the datetime of the reading, the total volume and the difference since the previous meter reading. The difference field value sometimes is inconsistent e.g. it is not equal to the difference between the volume values of two consecutive readings. To that end, the import process, always computes the value of the difference field. For rows, which there is no previous reading i.e. the first row for a serial number, HBase is queried to retrieve the most recent volume value.

4.1.5. Smart Water Meter Data Import from Filesystem

The uploaded SWM meter data is not always complete. By means of completeness, we mean that there may be missing SWM readings from a few hours to entire days. Moreover, sometimes the Smart Water Meter Data Import process may fail to parse and skip an input file altogether. Since the files remain on the SFTP server only for ten days, it is possible to miss some files. For this reason, a simplified version of the Smart Water Meter Data Import process was implemented that skips the execution of the first two tasks components. Instead, the working directory is set explicitly as a job parameter and the import task starts immediately. Using this process, any missing files can be copied to the DAIAD server and loaded in HBase on demand. This process is used mostly for batch loading of SWM data over long time intervals.

---

[19] https://app.dev.daiad.eu/docs/api/index.html#api-Weather

4.1.6. Flink Forecasting Job

In DAIAD we have streamlined the execution of Flink data analysis jobs by adhering to the following conventions:

- Every implemented Flink job is packaged as a single jar that contains all dependencies.
- A list of one or more arguments may be passed to the job from the command line.
- The last argument always refers to a HDFS working directory path. Any required input files must be stored in this directory before the Flink job execution begins. The job may save intermediate files and results in this directory.

Executing the Flink Forecasting job involves the following tasks:

- A temporary directory is created in the host which submits the Flink job.
- A temporary directory is created in HDFS filesystem and is used as the working directory for the Flink job.
- User water consumption data is exported in the local working directory. The exported file has four fields per row, namely, the SWM serial number, the datetime of the reading, the volume and the difference from the previous most recent reading.
- A forecasting query input file is generated. This file has the same format as the input file with two distinct differences, first the datetime refers to a future date for which we want to compute an estimate and second the volume and difference field values are just placeholders.
- A set of python scripts is executed that pre-process the input files and generates additional input files required by the algorithm.
- All the input files are copied from the local working directory to the remote working directory on HDFS.
- The Flink job is invoked and the HDFS working directory is passed as an argument.
- The generated result file is imported to HBase.
- The working directories are deleted.

The tasks are orchestrated by a custom Job Builder. Using a combination of these tasks, we can easily implement any custom Job Builder for executing other Flink jobs.

4.1.7. Data Pre-Aggregation MapReduce Job

Like Flink jobs, MapReduce job execution is streamlined by adhering to the following conventions:

- Every implemented MapReduce job is packaged either as a single “fat” jar that contains all dependencies or a jar with only the job implementation and an additional path that contains any required libraries. The parameter for this path is “mapreduce.job.lib.local”.

- A path may be passed as a job parameter and its contents will be copied to the HDFS and will be available during the job execution. The parameter name is “mapreduce.job.files.local”.
- A path may be passed as a job parameter and its contents will be cached on every data node and be available to every mapper and reducer using symbolic links on the data node’s host filesystem. The parameter name is “mapreduce.job.cache.local”.

Executing the Flink Forecasting job involves the following tasks:

- A temporary directory is created in HDFS filesystem and is used as the working directory for the MapReduce job.
- A temporary directory is created in HDFS filesystem and is used for storing any required libraries.
- If parameter “mapreduce.job.lib.local” is declared, the contents of the directory are copied to the HDFS libraries directory.
- If parameter “mapreduce.job.files.local” is declared, the contents of the directory are copied to the HDFS input files directory.
- If parameter “mapreduce.job.cache.local” is declared the contents of the directory are cached to every data node.
- The MapReduce job is submitted to the YARN cluster. The mappers group SWM data based on the serial number and datetime and the reducer aggregates group values and stores the result to HBase.
- The working directories are deleted.

The tasks are orchestrated by a custom Job Builder. If no extra actions are required after the MapReduce job execution, the same Job Builder can be used for running any MapReduce job simply by providing the appropriate parameters.

4.1.8. Amphiro b1 Schema Migration

During the development of the DAIAD system we had to revise the amphiro b1 schema several times as described in section 3.2. To simplify the data migration process, we implemented each schema migration as a separate task that transformed data form a previous version to the next newer one. Moreover, to manage DAIAD@home clients who uploaded data using older versions of the mobile application we created a cumulative ETL process. The corresponding job builder executes all registered migrations in reverse order, starting from the most recent migration to the least recent one. During the execution of a migration, all data from the legacy HBase table is retrieved and reimported in the current table using the Data API. The latter guarantees that all validation checks are applied and no inconsistencies are introduced.

4.1.9. Utility and Trial Data Export

In DAIAD we collect data mostly from two sources, namely, amphiro b1 data from the trials in Alicante and St. Albans and smart water meter data provided by AMAEM. The collected data is stored in HBase and is used by the data analysis and forecasting algorithms implemented in WP5. Moreover, this data must be available for use by external, 3rd party data analysis tools and programming languages like R[20]. To this end, an ETL job that exports all data stored in HBase has been implemented.

The ETL process creates multiple CSV text files that are eventually added into a single archive. In addition to the water consumption data files, several auxiliary files are also created that contain information about the trial phase timeline and errors that are detected during the execution. The process creates two types of archives, namely, amphiro b1 data and smart water data. For amphiro b1 data, a set of rules is applied for cleaning data. Data with too low flow, volume or duration is removed. Moreover, if too many showers are rejected for a single amphiro b1 device, an error is generated and logged. On the contrary, smart water meter data is exported without any additional validation since all data is cleaned during the execution of the import ETL process.

The amphiro b1 archive consists of the following files:

- **error.csv**: Contains error messages generated by the system during the execution of the ETL job. Each row consists of the unique user key, the user name, the unique device key and an error description.
- **user.csv**: Lists all the users who participate in the trial. The file columns are the unique user key and the user name.
- **phase-timestamp.csv**: This file contains information about the trial phases for each amphiro b1 device alongside with the corresponding time intervals. Each line contains the following four phases (a) Baseline (Mobile Off / amphiro b1 Off) both the mobile application and amphiro b1 display are disabled, (b) Phase 1 where either the mobile application or the amphiro b1 display is enabled, (c) Phase 2 where both the mobile application and amphiro b1 display are enabled and (d) Phase 3 where social features are enabled. It is not required that all phases are present e.g. a user may have multiple features enabled from the beginning of the trial.
- **phase-shower-id.csv**: As above, this file contains information about the trial phases for each user alongside with the corresponding shower id intervals. Nevertheless, depending on how often real-time shower data is uploaded, some phase intervals may not be present.
- **shower-data-all.csv**: Contains all the shower data. No filtering is applied. Except of shower properties such as duration, volume and energy, each row contains two additional Boolean properties, namely, history and ignore. The first is True for historical showers and False for real-time ones. The second is True only for showers that the user has characterized as not being showers.
- **shower-data-valid.csv**: Contains the shower data that has passed all validation rules.
- **shower-data-removed.csv**: Contains shower data that has failed to pass any of the validation rules.
- **shower-data-removed-index.csv**: Contains the indexes of the showers that have failed to pass any of the validation rules for each amphiro b1 device.
- **shower-time-series**: This file contains the time-series for the amphiro b1 real-time showers.

The smart water meter data archive contains the following 3 files:

---

[20] https://www.r-project.org/about.html

- **user.csv**: Lists all the users who participate in the trial together with the serial number of the smart water meter assigned to them.
- **data.csv**: Contains smart water meter data from all meters assigned to users who participate in the DAIAD trial. The file contains four columns, namely, meter id, local date time (e.g. Europe/Madrid time zone for Alicante), volume and difference (from the previous reading).
- **phase-timestamp.csv**: This file contains information about the trial phases for each user alongside with the corresponding time intervals.

4.2. Data API

The Data API supports querying data persisted by the Big Water Data Management engine. It is exposed as a HTTP RPC API that exchanges JSON encoded messages and has two endpoints, namely, the Action API and HTTP API endpoints. The former is a stateful API that is consumed by the DAIAD web applications. The latter is a CORS enabled stateless API that can be used by 3rd party applications.

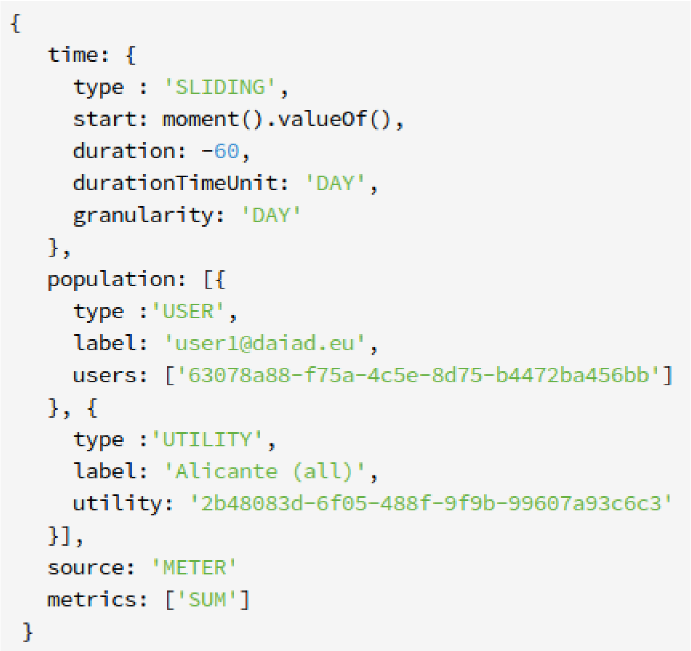
The API exposes data from three data sources, namely, smart water meter data, Amphiro b1 data and forecasting data for smart water meters. The query syntax is common for all data sources. Moreover, smart water meter and Amphiro b1 data can be queried simultaneously. However, a separate request must be executed for forecasting data.

The API accepts a set of filtering criteria as parameters and returns one or more data series consisting of data points which in turn have one or more aggregate metrics like sum, min or average values. More specifically the input parameters are:

- **Time**: Queries data for a specific time interval. An absolute time interval or a relative one (sliding window) can be defined. Optionally, the time granularity i.e. hour, day, week, month or year, can be declared that further partitions the time interval in multiple intervals. The Data API returns results for every of these time intervals.
- **Population**: Specifies one or more groups of users to query. For every user group a new data series of aggregated data is returned. A query may request data for all the users of a utility, the users of a cluster, the users of an existing group, a set of specific users or just a single user. Clusters are expanded to segments before executing the query. A segment is equivalent to a group of users. Thus, declaring a cluster is equivalent to declaring all its groups. Optionally, the users of any group may be ranked based on a metric.
- **Spatial**: A query may optionally declare one or more spatial constraints and filters. A spatial constraint aggregates data only for users whose location satisfies the spatial constraint e.g. it is inside a specific area. On the contrary, a spatial filter is like the population parameter and creates a group of users based on their location; hence a new data series is returned for every spatial filter.
- **Metric**: The metrics returned by the query. Data API supports min, max, sum, count and average aggregate operations. Not all data sources support all metrics.

- **Source**: Declares the data source to use. When data forecasting is requested, this parameter is ignored.

A simple query in JSON format is displayed in Figure 14. The specific query requests a single metric, namely SUM, for every day for the last 60 days. The result contains two data series, one for a single user named user1@daiad.eu and one for the entire population of the Alicante utility.



```
{
   time: {
     type : 'SLIDING',
     start: moment().valueOf(),
     duration: -60,
     durationTimeUnit: 'DAY',
     granularity: 'DAY'
   },
   population: [{
     type :'USER',
     label: 'user1@daiad.eu',
     users: ['63078a88-f75a-4c5e-8d75-b4472ba456bb']
   }, {
     type :'UTILITY',
     label: 'Alicante (all)',
     utility: '2b48083d-6f05-488f-9f9b-99607a93c6c3'
   }],
   source: 'METER'
   metrics: ['SUM']
 }
```

Figure 14: Data API request query example

Detailed documentation on the Data API syntax and request examples can be found at https://app.dev.daiad.eu/docs/api/index.html#api-Data-DataQuery.

The Data API is implemented as part of the DAIAD Services presented in Deliverable 1.3. Figure 15 illustrates the Data API implementation in more detail. Query requests are received by the DAIAD Services controller components and forwarded to the Data Service. The Data Service orchestrates data query execution. It accesses data from several repositories such as user profile information and smart water meter registration data and expands the query before execution. Query expansion refers to the process that selects all individual users and their corresponding devices for all groups to query. In addition, any spatial constraints are applied at this stage. The expanded query is either submitted as a Map Reduce job to the YARN cluster or executed against the HBase data tables described in section 3.2.

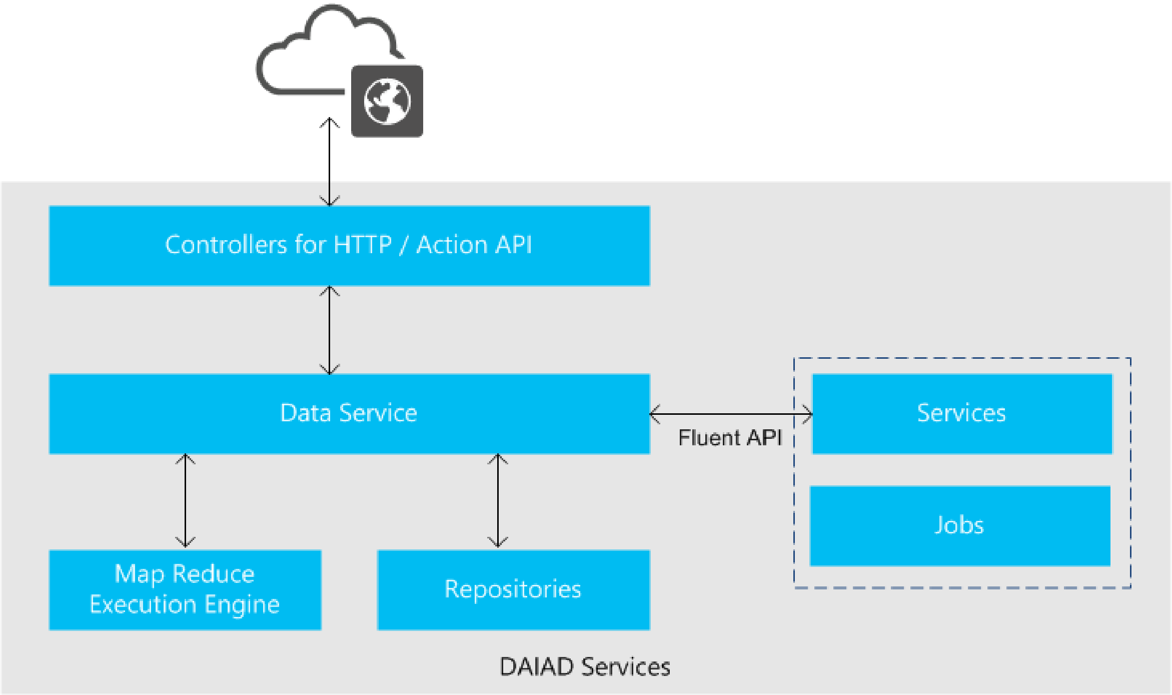

Controllers for HTTP / Action API
Data Service
Fluent API
Services
Jobs
Map Reduce Execution Engine
Repositories
DAIAD Services

Figure 15: Data API

In addition to the HTTP endpoints, Data API also provides a fluent API for building queries at the server side. This feature is used by other services and jobs for querying water consumption data. Two distinctive examples are the Message Service[21] and the User Clustering Job[22] respectively. The former queries utility and individual user consumption data to generate alerts and recommendations. The latter clusters the users based on their total water consumption over a predefined time interval.

---

[21] https://github.com/DAIAD/home-web/blob/master/src/main/java/eu/daiad/web/service/message/DefaultMessageService.java

[22] https://github.com/DAIAD/home-web/blob/master/src/main/java/eu/daiad/web/job/builder/ConsumptionClusterJobBuilder.java

5. Annex I: HBase row schema

DAIAD is using HBase to store water consumption data generated by smart water meters, amphiro b1 devices, and analysis algorithms. For raw smart water meter and amphiro b1 data the application maintains two copies. One is optimized for querying data for a single user/device (i.e. consumer) and the other for querying data over a time interval. In the next sections we present the HBase tables by displaying the parts that compose their row keys. All tables belong to the *daiad* namespace (not included in the table names for brevity).

5.1. amphiro-showers-by-time-v3

The row key format for amphiro b1 data is show in Figure 16. Contrary to the smart water meter data row key format, a single shower is stored per row key. Thus, column qualifiers are not prefixed by a timestamp offset.

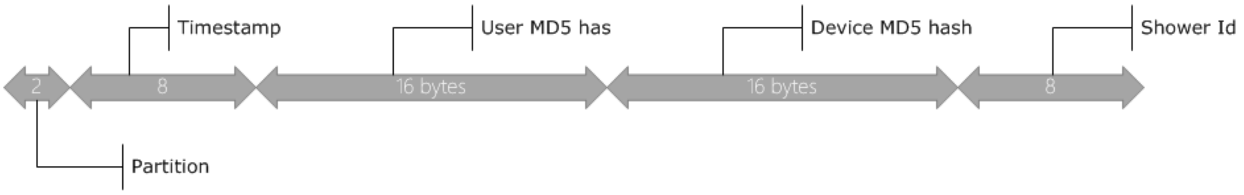


Figure 16: Time indexed amphiro b1 row key

The cells stored for every row are enumerated in Table 1. Not every cell value is present. More over the qualifier names are indicative. The application uses shorter aliases for column qualifiers in order to save storage space since in HBase the column qualifier is saved for every cell.

Column qualifier	Description
Historical shower timestamp	Unix timestamp[23] of the historical shower
Historical shower volume	Total volume in litres
Historical shower energy	Total energy in watts
Historical shower duration	-
Historical shower temperature	Average temperature in °C
Historical shower flow	Average flow in litres per minute
Real-time shower timestamp	Unix timestamp of the real-tome shower
Real-time shower volume	Total volume in litres
Real-time shower energy	Total energy in watts
Real-time shower duration	-

---

[23] https://en.wikipedia.org/wiki/Unix_time

Real-time shower temperature	Average temperature in °C
Real-time shower flow	Average flow in litres per minute
Member index	Household member index
Member mode	AUTO: If the application has assigned a member automatically MANUAL: if the user has selected a household member explicitly
Member timestamp	Unix timestamp when the member was assigned
Ignore	True if this is not a shower
Ignore timestamp	Unix timestamp when the shower was marked as ignored

Table 1: Amphiro b1 row cells

5.2. amphiro-showers-by-user-v3

Like amphiro-showers-by-time-v3 table, the Amphiro-showers-by-user-v3 table stores amphiro b1 showers. However, in this case the row key is optimized for querying data for a single user. The corresponding row key structure is the same as the one shown in Figure 17. For each row, the timestamp, volume, energy, temperature, flow and duration values are saved.

5.3. amphiro-measurements-v3

Table amphiro-measurements-v3 stores time-series for real-time showers generated by amphiro b1 devices. The row key structure is shown in Figure 17 and is optimized for searching time series for a specific user and device. A surrogate key is generated for column qualifiers that consists of the ordinal position of a measurement and the field name. Hence, a physical HBase row contains all logical rows for all measurements of a single shower.

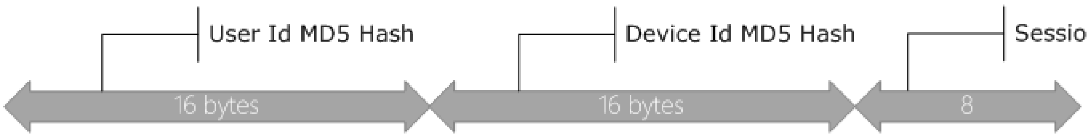


Figure 17: Table amphiro-measurements-v2 row key structure

5.4. meter-measurements-by-time

Table meter-measurements-by-time stores smart water meter measurements and is optimized for searching data for multiple users over a time interval. For every measurement the volume and difference since the last measurement are stored. The row key structure is shown in Figure 18.

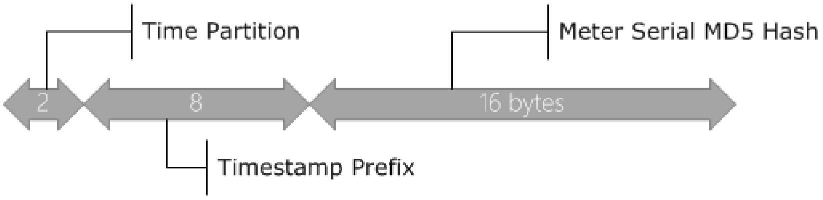


Figure 18: Table meter-measurements-by-time row key structure

5.5. meter-measurements-by-user

Table meter-measurements-by-user stores smart water meter measurements and is optimized for searching data for a single smart water meter device. For every measurement the volume and difference since the last measurement are stored. The row key structure is displayed in Figure 19.

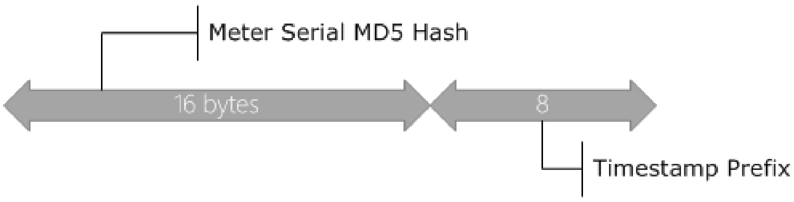


Figure 19: Table meter-measurements-by-user row key structure

5.6. meter-forecast-by-time

Table meter-forecast-by-time is similar to the meter-measurements-by-time table but instead of storing actual measurement data, it stores the results of the forecasting algorithms. The row key has the same structure as in section 5.4.

5.7. meter-forecast-by-user

Table meter-forecast-by-user stores smart water meter data generated by the forecasting algorithms. Its row key has the same structure as in section 5.5 and hence it is optimized for querying the data of a single meter

5.8. User efficiency ranking

DAIAD@utility periodically executes a data analysis job that creates clusters of users based on their water consumption efficiency. The results of the job are stored in a HBase table and are available for querying to the end users of DAIAD@home applications. The row key for this table is shown in Figure 20and the corresponding cells in Table 2.

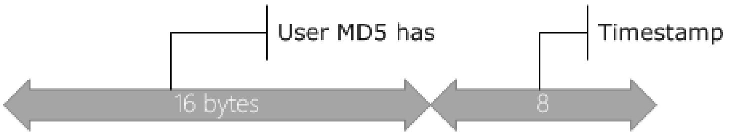


Figure 20: Table user efficiency ranking row key structure

Column Qualifier	Description
User	User total consumption.
Similar	Total consumption for similar users.
Nearest	Total consumption for users who are geographically near to the user
Utility	Total consumption for the whole utility
Week	Week of year

Table 2: Ranking row cells

6. Annex II: PostgreSQL table schema

In DAIAD we use PostgreSQL for storing frequently changing data, like user accounts and profiles, smart water meter and amphiro b1 registrations, application metadata, cluster management information, etc. In general, PostgreSQL stores tables that contain only a few hundreds of thousands of records. The goal of using a relational database for this type of data is to minimize update and delete operations to HBase.

DAIAD separates data into two database instances. One instance stores user specific information like user accounts and roles. The other one stores system specific information such as scheduled job metadata. In the next sections we describe the most important tables for every database. The complete schema for both databases expressed in SQL Data Definition Language (DDL) is available at the DAIAD repository[24]. Tables that are generated automatically by 3rd party libraries (*Flyway schema migration tool and Spring Batch Job Repository*), are not included in this presentation.

6.1. System Database

System database instance is used by the scheduler service for storing data for:

- Job schedule configuration
- Job execution parameters
- Application statistics
- Information about job execution results e.g. uploaded file metadata

6.1.1. daily_counter

Stores daily statistics about the DAIAD application runtime. Daily statistics are computed by the job configured in DailyStatsCollectionJobBuilder[25] class.

Field	Name
id	Unique Id
utility_id	Reference to utility table
date_created	Row creation date
counter_name	Counter name
counter_value	Counter value

---

[24] https://github.com/DAIAD/home-web/tree/master/src/main/resources/db/migration

[25] https://github.com/DAIAD/home-web/blob/master/src/main/java/eu/daiad/web/job/builder/DailyStatsCollectionJobBuilder.java

6.1.2. scheduled_job

Stores configuration data for scheduler service registered jobs. DAIAD application consults this table about what jobs to schedule and how to create instances of these jobs. Users can schedule jobs for periodic execution based on a fixed interval or a CRON expression. Only one of these options must be specified. Setting a value for both fields will result in a runtime error.

Field	Name
id	Unique job registration Id.
category	Job category. Valid values are: MAINTENANCE ETL ANALYSIS FORECASTING
container	Execution framework. Valid values are: RUNTIME HADOOP FLINK
bean	The name of the class that implements IJobBuilder interface for the specific job. DAIAD application uses this class for creating an instance of the job before execution.
name	Unique name of the job. This name is used internally by DAIAD in order to identify a job.
description	User friendly job description.
data_created	Row creation date.
period	Interval in seconds for scheduling the job for periodic execution.
cron_expression	A valid CRON expression for scheduling the job.
enabled	True if the job must be scheduled; Otherwise False.
visible	True if the job is listed in the DAIAD@utility job management view.

6.1.3. scheduled_job_parameter

Stores execution parameters for a registered job. Parameters in this table are passed to the job context before job execution by the scheduler service.

Field	Name
id	Unique Id.
scheduled_job_id	Reference to scheduled job table.
name	Parameter name.
value	Parameter value.
hidden	True if the parameter should not be published by the Action API; Otherwise False.
step	The name of the step that uses this parameter.

6.1.4. upload

Stores metadata for files downloaded by the job configured by WaterMeterDataSftpLoadJobBuilder[26] class.

Field	Name
id	Unique Id
source	Remote SFTP server address.
remote_folder	Remote folder.
local_folder	Local folder.
remote_filename	Remote file name.
local_filename	Local file name.
file_size	Remote file size.
date_modified	File last modified date.
upload_start_on	Timestamp of download operation start.
upload_end_on	Timestamp of download operation end.
process_start_on	Timestamp of data process operation start.
process_end_on	Timestamp of data process operation end.
row_count	Total number of rows in the file.
row_processed	Number of rows processed.
row_skipped	Number of rows skipped due to validation errors.

---

[26] https://github.com/DAIAD/home-web/blob/master/src/main/java/eu/daiad/web/job/builder/WaterMeterDataSftpLoadJobBuilder.java

row_negative_difference	Number of rows for which a negative variation in consumption has been found.

6.1.5. export

Stores metadata for the export data files created during the ETL process described in section 4.1.

Field	Name
id	Unique Id.
key	Unique key (UUID).
created_on	File creation date.
started_on	Export start datetime.
completed_on	Export end datetime.
path	Path where the file is stored.
filename	-
file_size	-
utility_id	-
utility_name	-
description	Short description of the export file.
row_count	Total number of rows in file if this is a simple text file or sum of all rows in all files if this is an archive.

6.2. Application Database

The application database is used for persisting DAIAD Domain Objects, which include:

- User accounts, roles and profiles
- Registrations for smart water meters and amphiro b1 devices
- Customized alerts, recommendations, tips and announcements for users
- User groups, clusters and cluster segments

6.2.1. account

Account, password and security settings.

Field	Name
id	-
row_version	Row version used by the Object Relation Mapper (ORM) for implementing optimistic row locking.
utility_id	Reference to utility table.
key	UUID used for generating MD5 hash values when computing HBase row keys.
locale	-
firstname	-
lastname	-
email	-
created_on	-
last_login_success	-
last_login_failure	-
failed_login_attempts	Number of failed login attempts since the last successful login operation.
change_password_on_login	True if user must change her password after the next successful login operation.
locked	-
username	Unique username. Current implementation requires the username to be the same as the user email field.
password	Encrypted password hash with salt.
photo	-
timezone	-
country	-
city	-
address	-
postal_code	-
birthdate	-
gender	-

location	User location expressed in WGS84 coordinates.

6.2.2. account_alert

Account specific alerts generated by the messaging module.

Field	Name
id	-
acknowledged_on	When the user has viewed the alert.
created_on	When the alert has been generated.
receive_acknowledged_on	When the DAIAD@utility received the confirmation that the alert has been viewed by the user.
account_id	Reference to account table.
alert_template	Reference to alert_template table.
device_type	Device type: amphiro b1 or smart water meter.
resolver_execution	(used internally for resolving implementation class)

6.2.3. account_ announcement

Recipients of utility announcements.

Field	Name
id	-
account_id	Reference to account table.
announcement_id	Reference to announcement table.
created_on	-
acknowledged_on	When the user has viewed the announcement.

6.2.4. account_profile

Generic account and application settings.

Field	Name
id	-
row_version	Row version used by the ORM for implementing optimistic row locking.

version	Profile internal version.
updated_on	-
mobile_mode	DAIAD@home mobile application mode.
mobile_config	DAIAD@home mobile application custom configuration options.
web_mode	DAIAD@home web application mode.
web_config	DAIAD@home web application custom configuration options.
utility_mode	DAIAD@utility web application mode.
utility_config	DAIAD@utility web application custom configuration options.
daily_meter_budget	Daily budget for smart water meter in liters.
daily_amphiro_budget	Daily budget for amphiro b1 in liters.
social_enabled	True if social features are enabled.
unit	Unit system used (Metric or Imperial).
send_mail	True if mails should be sent to the user.
mobile_app_version	Most recent mobile application version detected by the Data API.
send_message	True if message generation is enabled for this user.

6.2.5. account_profile_history

Log of all profile updates.

Field	Name
id	-
profile_id	Reference to account_profile table
version	Profile internal version.
updated_on	-
acknowledged_on	When the DAIAD@utility received the confirmation that the profile has been enabled at the client.
enabled_on	When the client has enabled the profile.
mobile_mode	DAIAD@home mobile application mode.
web_mode	DAIAD@home web application mode.
utility_mode	DAIAD@utility web application mode.
social_enabled	True if social features are enabled.

6.2.6. account_recommendation

Personalized recommendations generated by the messaging module.

Field	Name
id	-
acknowledged_on	When the user viewed the recommendation.
created_on	When the recommendation was generated.
receive_acknowledged_on	When the DAIAD@utility received the confirmation that the recommendation has been viewed by the user.
account_id	Reference to account table.
recommendation_template	Reference to recommendation_template table.
device_type	Device type: amphiro b1 or smart water meter.
resolver_execution	(used internally for resolving implementation class)
significant	Level of recommendation significance.

6.2.7. account_role

Role assignment to accounts.

Field	Name
id	-
account_id	Reference to account table.
role_id	Refence to role table.
date_assigned	-
assigned_by	-

6.2.8. account_tip

Account specific tips generated by the messaging module for reducing water consumption.

Field	Name
id	-
account_id	Reference to account table.
tip_id	Reference to tip table.
created_on	When the tip was generated.

acknowledged_on	When the user viewed the tip.
receive_acknowledged_on	When the DAIAD@utility received the confirmation that the tip has been viewed by the user.

6.2.9. account_white_list

Access control list for account registration.

Field	Name
id	-
utility_id	Reference to the utility table.
account_id	Reference to the account table if the user has been registered.
username	-
registered_on	-
locale	-
firstname	-
lastname	-
timezone	-
country	-
postal_code	-
birthdate	-
gender	-
default_mobile_mode	-
default_web_mode	-
city	-
address	-
row_version	-
meter_location	-
meter_serial	-
location	-

6.2.10. alert_template

Templates for generating alerts.

Field	Name
value	Unique Id.
name	Template name.
type	Template type.

6.2.11. alert_template_translation

Translations for alert templates.

Field	Name
id	-
template	Reference to template table.
locale	-
description	Full description.
title	Short description.
link	Link to external resource.

6.2.12. announcement

Utility announcements.

Field	Name
id	-
priority	-

6.2.13. announcement_translation

Announcement translations.

Field	Name
id	-
announcement_id	Reference to announcement table.
locale	-
title	-

content	-
link	Link to external content.

6.2.14. area_group

Groups of spatial areas e.g. municipal administrative boundaries.

Field	Name
id	-
utility_id	Reference to utility table.
key	UUID used for generating MD5 hash values when computing HBase row keys.
row_version	Row version used by the ORM for implementing optimistic row locking.
title	-
created_on	-
bbox	Bounding box.
level_count	Number of levels.

6.2.15. area_group_item

A spatial area grouped under a group area.

Field	Name
id	-
utility_id	Reference to utility table.
area_group_id	Reference to area_group table.
key	UUID used for generating MD5 hash values when computing HBase row keys.
row_version	Row version used by the ORM for implementing optimistic row locking.
title	-
created_on	-
the_geom	-

level_index	Zero-based level index. Index must be less than area_group.level_count. If area_group_id is null, this field is ignored.

6.2.16. cluster

Clusters of users generated by an analysis job.

Field	Name
id	Unique Id.
key	UUID used for generating MD5 hash values when computing HBase row keys.
row_version	Row version used by the ORM for implementing optimistic row locking.
utility_id	Reference to utility table.
name	Unique name.
created_on	Row creation date.

6.2.17. data_query

Custom queries for DAIAD@utility, executable by the Data API.

Field	Name
id	-
account_id	Reference to account table.
name	User friendly name for the query.
query	The query serialized in JSON format.
date_modified	-
type	-
tags	-
report_name	-
field	-
time_level	-
overlap	-
pinned	True if the query is displayed in the DAIAD@utility dashboard.

6.2.18. device

Basic registration data for smart water meter and amphiro b1 devices.

Field	Name
id	Unique numeric id of the device.
key	UUID used for generating MD5 hash values when computing HBase row keys.
row_version	Row version used by the ORM for implementing optimistic row locking.
account_id	-
registered_on	-
last_upload_success_on	Last successful data uploading operation timestamp.
last_upload_failure_on	Last failed data uploading operation timestamp.
transmission_count	Number of data transmissions.
transmission_interval_sum	Sum of all data transmission intervals.
transmission_interval_max	Maximum interval between two successive data transmissions.

6.2.19. device_amphiro

Amphiro b1 specific registration data.

Field	Name
id	-
name	User friendly name.
mac_address	-
aes_key	AES key used for data encryption.

6.2.20. device_meter

Smart water meter specific registration data.

Field	Name
id	-
serial	Unique meter serial number.

location	-

6.2.21. device_property

Device arbitrary data represented as key / value pairs of string literals.

Field	Name
id	-
device_id	Reference to device table.
key	Unique per device property key.
value	-

6.2.22. favourite

References to selected users or groups.

Field	Name
id	-
key	UUID used for generating MD5 hash values when computing HBase row keys.
row_version	Row version used by the ORM for implementing optimistic row locking.
owner_id	Reference to account table.
label	User friendly name.
created_on	-

6.2.23. group

Groups of users. A group may be a custom set, a cluster segment or a DAIAD@commons group.

Field	Name
id	Unique numeric id of the group.
key	UUID used for generating MD5 hash values when computing HBase row keys.
row_version	Row version used by the ORM for implementing optimistic row locking.
utility_id	-

name	-
created_on	-
spatial	Group geometry representation in WGS84.
size	Number of group members.
updated_on	-

6.2.24. group_commons

Data specific to DAIAD@commons groups.

Field	Name
id	-
description	-
image	-
owner_id	Reference to account table.

6.2.25. group_member

Table for storing user memberships to groups. A user may belong to more than one groups.

Field	Name
id	-
group_id	Reference to group table.
account_id	Reference to account table.
created_on	-

6.2.26. group_segment

Data specific to a cluster segment.

Field	Name
id	-
cluster_id	Reference to cluster table.

6.2.27. group_set

Data specific to a custom set of users.

Field	Name
id	-
owner_id	Reference to account table.

6.2.28. household

User household information.

Field	Name
id	-
row_version	Row version used by the ORM for implementing optimistic row locking.
created_on	-
updated_on	-

6.2.29. household_member

Household members.

Field	Name
id	-
row_version	Row version used by the ORM for implementing optimistic row locking.
household_id	Reference to household table.
index	Ordinal position.
name	-
age	-
gender	-
photo	-
created_on	-
updated_on	-
active	True if the member is active. Household members are never deleted. Instead, active field is set to false.

6.2.30. log4j_message

Application log for easy access to the log files from the application User Interface (UI).

Field	Name
id	Unique Id.
account	Principal name i.e. the name of the authenticated account if any existed when the message was being logged.
remote_address	Client remote address.
category	Message category e.g. scheduler, data query etc.
code	Error specific code e.g. authentication missing credentials error.
level	Error level e.g. information, debug, warning, critical etc.
logger	Class that logged the message.
message	Message.
exception	Detailed exception message.
timestamp	When message has been logged.
Field	Name

6.2.31. password_reset_token

Transient unique tokens used for resetting passwords.

Field	Name
id	-
account_id	Reference to account table.
created_on	-
redeemed_on	-
token	-
valid	True if the token has not been redeemed or expired.
pin	Unique number sent to user.
application	Application requesting the password reset token e.g. DAIAD@home or DAIAD@utility.

6.2.32. recommendation_template

Templates for generating recommendations.

Field	Name
value	Unique id.
name	-
type	-

6.2.33. recommendation_template_translation

Translations for recommendation templates.

Field	Name
id	-
template	Reference to recommendation_template table.
locale	
description	-
title	-
image_link	-

6.2.34. role

Application roles such as user, administrator etc.

Field	Name
id	-
name	-
description	-

6.2.35. tip

Tips for reducing water consumption.

Field	Name
id	-
index	Ordinal position in category.
category_id	-

locale	-
title	-
description	-
image_binary	Base64 encoded image.
image_link	-
prompt	-
externa_link	-
created_on	-
modified_on	-
active	True if the tip can be selected by the message generator.
source	-
image_mime_type	-

6.2.36. utility

Utility generic data and default values for utility accounts.

Field	Name
id	Unique numeric id of the utility.
key	UUID used for generating MD5 hash values when computing HBase row keys.
name	-
logo	-
description	-
date_created	Row creation datetime.
default_admin_username	Whenever a new utility is created, the application initializes a new administrator account for it with the username from this field.
locale	-
timezone	-
country	-
city	-

default_amphiro_mode	Amphiro b1 mode for new users. Default mode is LEARNING
default_mobile_mode	Mobile application mode for new users. Default mode is LEARNING
default_web_mode	DAIAD@home web application mode for new users. Default mode is INACTIVE
default_social_mode	Social features mode for new users. Default mode is INACTIVE.
message_generation_enabled	True if messages such as alerts, recommendations and tips must be generated. By default, message generation is disabled.

6.2.37. water_iq_history

Comparison and ranking data generated by water consumption data analysis algorithms.

Field	Name
id	-
account_id	Reference to account table.
created_on	-
interval_from	Start of monthly interval.
interval_to	End if monthly interval.
user_volume	User total consumption.
user_value	User ranking value e.g. A, B, C, etc.
similar_volume	Average total consumption for similar users.
similar_value	Average ranking value for similar users.
nearest_volume	Average total consumption for geographically near users.
nearest_value	Average ranking value for geographically near users.
all_volume	Average total consumption for all users.
all_value	Average ranking for all users.
user_1m_consumption	Total user consumption.
similar_1m_consumption	Total consumption for similar users.
nearest_1m_consumption	Total consumption for geographically near users.
all_1m_consumption	Total consumption of all users.
interval_year	-
interval_month	-

6.2.38. weather_data_day

Daily weather data.

Field	Name
id	-
utility_id	Reference to utility table.
service_id	Reference to weather_service table.
min_temperature	-
max_temperature	-
min_temperature_feel	-
max_temperature_feel	-
min_humidity	-
max_humidity	-
precipitation	-
wind_speed	-
wind_direction	-
conditions	Description of weather conditions.
created_on	-
date	-

6.2.39. weather_data_hour

Hourly weather data.

Field	Name
id	
day_id	Reference to weather_data_day table.
temperature	-
temperature_feel	-
humidity	-
precipitation	-
wind_speed	-

wind_direction	-
conditions	Description of weather conditions.
datetime	-

6.2.40. weather_service

Weather services for harvesting meteorological data.

Field	Name
id	-
name	-
description	-
endpoint	API endpoint.
website	Service web site.
registered_on	-
bean	Class implementing the service.
active	True if the service should be queried when harvesting weather data.

6.2.41. survey

Contains survey data for users participating in DAIAD trials. Information from this table is used for generating user clusters based on their demographic characteristics.

Field	Name
username	-
firstname	-
lastname	-
address	-
city	-
gender	-
number_of_showers	Number of showers in the household.
smart_phone_os	-
tablet_os	-
apartment_size_bracket	-

age	-
household_member_total	Total number of members in the household.
household_member_female	Number of female members in the household.
household_member_male	Number of male members in the household.
income_bracket	-
meter_id	Smart water meter unique id. If this field has a value, a meter is assigned to a new account once registered.
utility_id	-
shower_per_week	Average number of showers per week.