REPORT ON DELIVERABLE 5.2.2

Updated Consumption Analytics and Forecasting 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 40, 28/06/2017
Actual Submission Date	28/06/2017
Work Package	WP5 Big Water Data Analysis
Task	Task 5.2 Consumption Analytics Task 5.3 Scalable Forecasting and What-if Analysis
Type	Prototype
Approval Status	Submitted for approval
Version	1.0
Number of Pages	82
Filename	D5.2.2_Updated_Consumption_Analytics_and_Forecasting_Engine.pdf

Abstract

This report presents an overview of the Prototype Deliverable D5.2.2 "Updated Consumption Analytics and Forecasting Engine", which includes all DAIAD software components developed in the context of Tasks 5.2 "Consumption Analytics" and 5.3 "Scalable Forecasting and What-if Analysis". First, we provide an overview of the processes and the architecture of the engine. Then, we present our FML algorithmic framework, consisted by three machine learning algorithms (FML-kNN, FML-SP and FML-RF), which support predictive and other analytics for water consumption on a city-scale. Further, we introduce the BTSR-Tree hybrid index, our novel indexing method tailored for efficiently answering hybrid queries, consisted of spatial coordinates and time series data. Finally, we present the experimental evaluation and benchmarking of our work.

History

version	date	reason	revised by
0.1	29/03/2017	First draft	Spiros Athanasiou
0.2	14/04/2017	Revisions in all sections	Giorgos Giannopoulos, Yannis Kouvaras
0.5	02/06/2017	Revisions in all sections	Pantelis Chronis, Giorgos Chatzigeorgakidis, Spiros Athanasiou, Yannis Kouvaras, Michalis Alexakis
0.6	04/06/2017	Updated figures and references	Pantelis Chronis, Giorgos Chatzigeorgakidis
0.7	10/06/2017	Minor edits and revisions	Yannis Kouvaras
0.9	24/06/2017	Revisions in all sections	Pantelis Chronis, Giorgos Chatzigeorgakidis, Spiros Athanasiou, Yannis Kouvaras, Michalis Alexakis
1.0	28/06/2017	Final version	Spiros Athanasiou

Author list

organization	name	contact information
ATHENA RC	Spiros Athanasiou	spathan@imis.athena-innovation.gr
ATHENA RC	Giorgos Giannopoulos	giann@imis.athena-innovation.gr
ATHENA RC	Yannis Kouvaras	jkouvar@imis.athena-innovation.gr
ATHENA RC	Giorgos Chatzigeorgakidis	gchatzi@imis.athena-innovation.gr
ATHENA RC	Michalis Alexakis	alexakis@imis.athena-innovation.gr
ATHENA RC	Pantelis Chronis	pchronis@imis.athena-innovation.gr

Executive Summary

This report presents an overview of the Prototype Deliverable D5.2.2 "Updated Consumption Analytics and Forecasting Engine", which includes all DAIAD software components developed in the context of Tasks 5.2 "Consumption Analytics" and 5.3 "Scalable Forecasting and What-if Analysis".

In Section 1, we present an overview of the *Scalable Analytics and Forecasting engine*, covering its major technology and implementation aspects. First, we revisit the *Big Data Engine* of DAIAD (D5.1.2) focusing on its *architecture* and the various data management and processing frameworks it integrates. As all analytics and forecasting algorithms are deployed *on top* of our engine, understanding its characteristics and capabilities is crucial. In the following, we describe how analytics tasks are managed, scheduled, and implemented across the different data processing frameworks of DAIAD. Finally, we present an overview of all implemented *analytics* and *forecasting* facilities, distinguishing between those implemented as *out-of-the-box* facilities (*i.e., queries on top of our engine*) vs. those delivered by novel algorithms developed for the project.

In Section 2, we present our work on developing the novel *FML algorithmic framework*, consisted of *three* machine learning algorithms, which have been *integrated* in the Consumption Analytics and Forecasting Engine to provide predictive analytics at the city-level. The first algorithm is *FML-kNN*, which supports two major machine learning processes, *classification* and *regression*. Contrary to similar approaches, FML-kNN is executed in a *single distributed session* achieving better time performance and operational efficiency as it eliminates costly operations, i.e., fetching and storing the intermediate results among the execution stages. *FML-SP* is the second algorithm of the FML algorithmic framework, which computes the *savings potential* of groups of households. The savings potential is an estimate of the household's *inelastic* consumption, i.e., the absolute minimum consumption required to sustain safe and healthy well-being. Finally, *FML-RF*, the third algorithm of the FML algorithmic framework, is a distributed *random forest* classification algorithm supporting the prediction of household and consumer characteristics from water use, as well as the identification of water demand determinants.

In Section 3, we introduce the *BTSR-Tree hybrid index,* an indexing method for *geolocated* time series data, i.e., time series annotated with location information. The consumption datasets we have collected in the context of DAIAD are such *hybrid* time series, since the location of the households is known. We, thus, are able to quickly and efficiently retrieve the results of various useful hybrid queries that involve *similarity searching* in both the spatial and time series domain. The index *prunes* the tree in both domains simultaneously, thus, significantly gaining in performance against similar baseline methods.

In Section 4, we present a comparative benchmarking evaluation of FML-kNN based on synthetic data. Further, we apply and evaluate FML-kNN against real-world data in two major cases for the project: (a) forecasting a households' water consumption *simultaneously* for all households within a city, (b) producing predictive analytics from shower events performed by *multiple* households. We also present a comparative evaluation of the BTSR-Tree hybrid index, comparing its performance on *five* different hybrid queries and for various parameter settings with a *baseline R-Tree* method, using *four* real-world datasets. The results demonstrate the advantages of our approach and establish it as a promising topic for our future research.

Abbreviations and Acronyms

aNN	Artificial Neural Networks
BTSR-Tree	Bundled Time Series R-Tree
CSV	Comma Separated Values
DAG	Directed Acyclic Graph
DTW	Dynamic Time Warping
F-*k*NN	Flink *k*-Nearest Neighbors
FML-*k*NN	Flink Machine Learning *k*-Nearest Neighbors
FML-RF	Flink Machine Learning Random Forest
FML-SP	Flink Machine Learning Savings Potential
H-z*k*NNJ	Hadoop z-order *k*-Nearest Neighbors Joins
HDFS	Hadoop Distributed File System
iSAX	indexed Symbolic Aggregate Approximation
*k*NN	*k* Nearest Neighbors
MBR	Minimum Bounding Rectangle
MBTS	Minimum Bounding Time Series
ML	Machine Learning
NYC	New York City
PAA	Piecewise Aggregate Approximation
PoI	Point of Interest
RMSE	Root Mean Squared Error
S-*k*NN	Spark *k*-Nearest Neighbors
SaaS	Software-as-a-Service

SAX	Symbolic Aggregate Approximation
SFC	Space Filling Curves
SWM	Smart Water Meter
TSR-Tree	Time Series R-Tree
UDF	User Defined Function
UK	United Kingdom
YARN	Yet Another Resource Negotiator

Table of Contents

1. Implementation

1.1. Overview

The Consumption Analytics and Forecasting Engine has been implemented over the Big Water Data Management engine (*for details, please refer to Prototype Deliverable D5.1.2*).

The engine consists of a software stack which includes the Hadoop Distributed File System (HDFS), offering a redundant storage environment and high throughput data access for huge volumes of data. The Apache HBase wide-column NoSQL database offers low latency, random data access over HDFS and can handle tables with billions of rows consisting of millions of columns. The YARN resource manager decouples data processing frameworks from resource management. The Hadoop MapReduce big data processing framework simplifies the development of highly parallelized and distributed programs. And finally, Apache Flink, a scalable data processing platform, lying over the HDFS, supports MapReduce-based operations and data transformations providing a mechanism for automatic procedure optimization that achieves better performance of iterative MapReduce algorithms compared to other platforms.

Figure 1 depicts the software stack of the DAIAD Big Data engine.

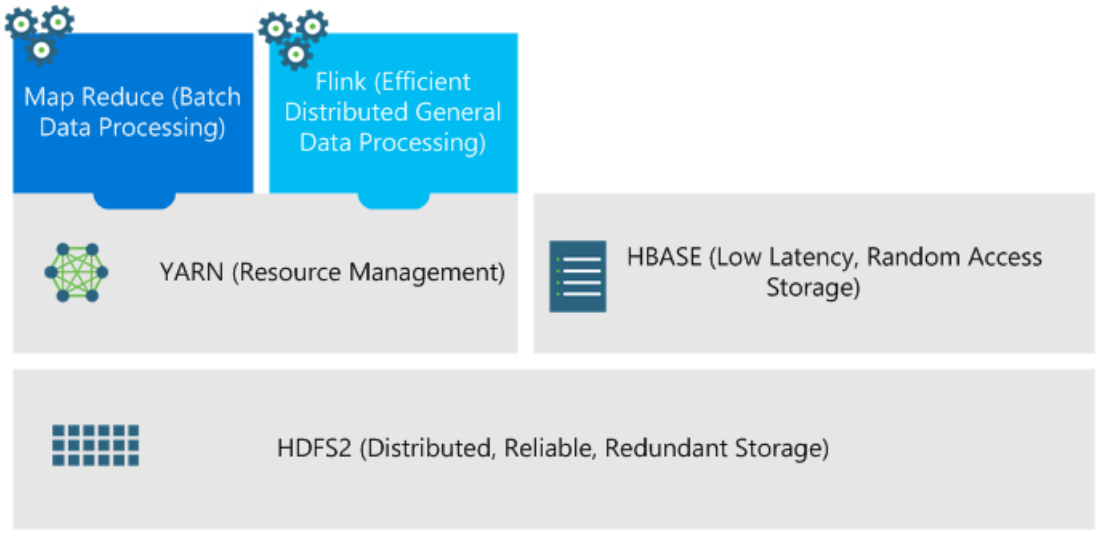


Figure 1: DAIAD Big Data Engine stack

1.2. Data engines

1.2.1. 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 2.

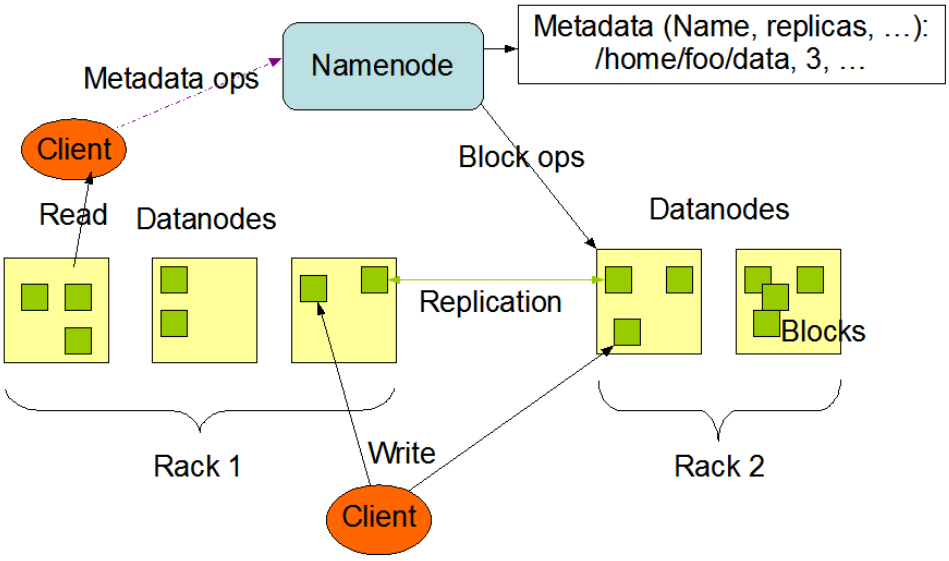


Figure 2: HDFS architecture

The NameNode is a single point of failure in a HDFS cluster. 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.

1.2.2. 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 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 the 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.

1.2.3. 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.

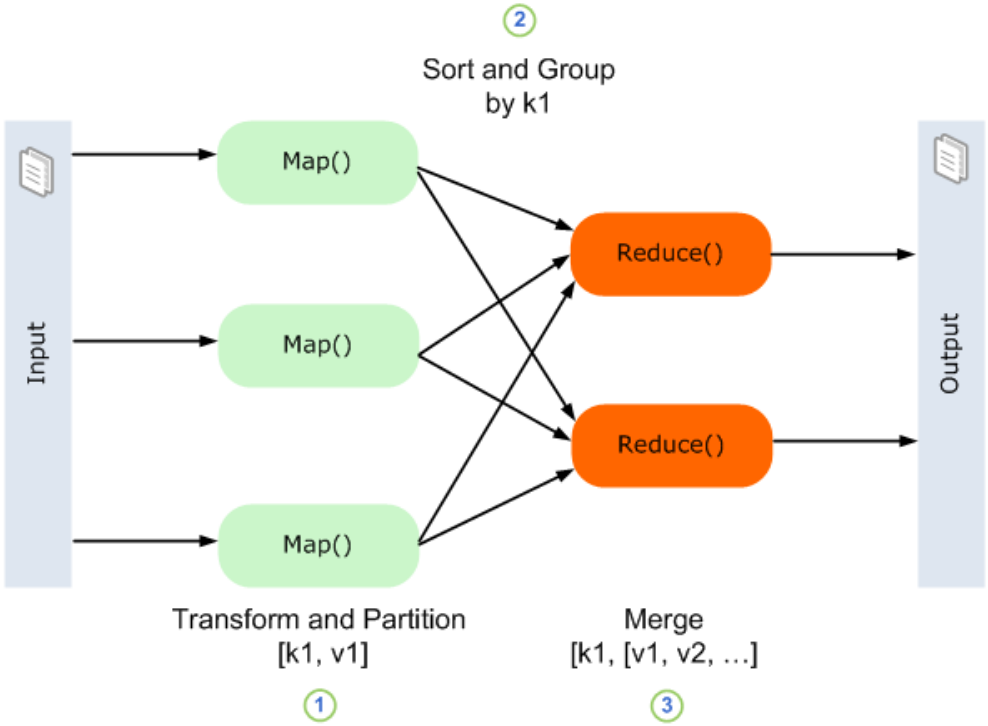


Figure 3: MapReduce computing model

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. 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.

The computing model of Hadoop Map Reduce is depicted in Figure 3. 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.

1.2.4. Apache Flink

Apache Flink is a data processing system for analyzing large datasets. Flink is not only built on top of an existing MapReduce framework, but also implements its own job execution runtime. Therefore, it can be used either as an *alternative* to Hadoop MapReduce platform 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 operations, called *transformations*. An operation consists of two components:

- A User-Defined Function (UDF). The function provided by the user.
- A parallel operator function. It parallelizes the execution of the UDF and applies the UDF on its input data.

The data model used by Flink operations is record-based while in MapReduce it is a *key-value* pair. Still, key-value pairs can be mapped to records. All operations are able to start in-memory executions and only when the resources become low they fall back to the external memory. The new operations efficiently support several data analysis tasks.

Flink allows to model job processing as DAGs of operations, which is more flexible than MapReduce, in which map operations are strictly followed by reduce operations. The combination of various operations allows for:

- Data pipelining. Transformations do not wait for preceding transformations to finish in order to start processing data.
- In-memory data transfer optimizations. Writing to disk is automatically avoided when possible.

Both these characteristics increase the performance, as they reduce disk access and network traffic.

Moreover, Flink efficiently supports *iterative algorithms*, which are important for Data Mining, Machine Learning and Graph exploration, since such processes often require iterating over the available data multiple times. In MapReduce, this becomes expensive, since data are transferred between iterations by using the distributed storage system. In the contrary, Flink supports the execution of iterative algorithms. Figure 4

illustrates Flink's pipelined execution and support of iterations. While the first Map transformation is running, a second source can be read and a second Map transformation can be initiated in parallel.

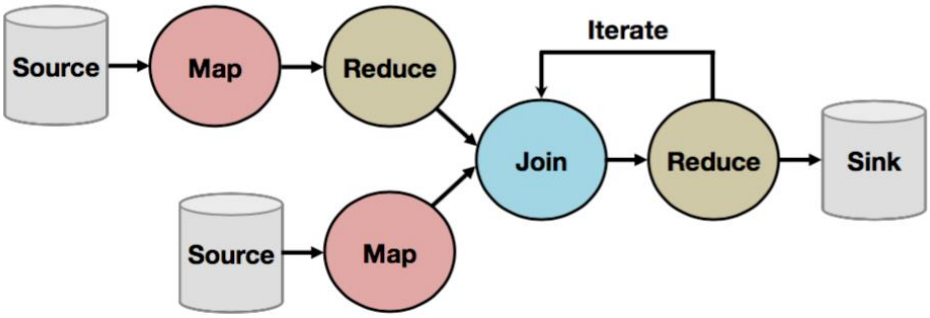


Figure 4: Flink's pipelined execution [BF14]

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

1.3. Execution

An overview of how jobs are initialized and submitted, as well as the processing frameworks used for job execution and how they fit in the Big Water Data Management Engine architecture, is presented in Figure 5. Job execution can be initialized either explicitly, by submitting a query to the Data Service, or manually through the Scheduler Service. In either case, the job is submitted to the Big Water Data Management Engine for execution.

Data and Scheduler Services are completely decoupled from Job classes implementation details and have no knowledge of the underlying processing framework used for execution. Information about the processing framework is encapsulated inside each Job class implementation.

The scheduler service is loosely coupled with Job classes instances through an appropriate interface that all Job classes must implement. This interface allows the service to configure Job instances with external parameters before submitting them for execution.

Job classes are separated into three categories depending on the processing framework they support:

- Simple Jobs: These jobs are implemented as simple Java classes and are executed in the same process at the application server that invokes them (*i.e., in-process execution*). They usually perform simple tasks, such as running SQL scripts or selecting data from the Big Water Data Management Engine for a single user or just a few users.
- Map Reduce Jobs: These jobs are executed using the Map Reduce processing framework, are accessing large volumes of data and usually perform simple aggregation and filtering to the data.

- Flink Jobs: These jobs are executed using the Flink processing framework, they are accessing large volumes of data and perform complex data analysis operations that may contain recursive operators. The latter cannot be implemented efficiently using MapReduce since they require chaining successive Map and Reduce operations.

Job submission to the Big Water Data Management Engine and the appropriate processing framework is handled by the Job class implementation. For simple jobs, both the initialization and implementation code are located inside the Job class. For Flink and MapReduce jobs, Jobs contain only the initialization logic and defer the actual implementation to external Java Archive (JAR) files that are loaded and invoked externally by the Flink and Map Reduce frameworks.

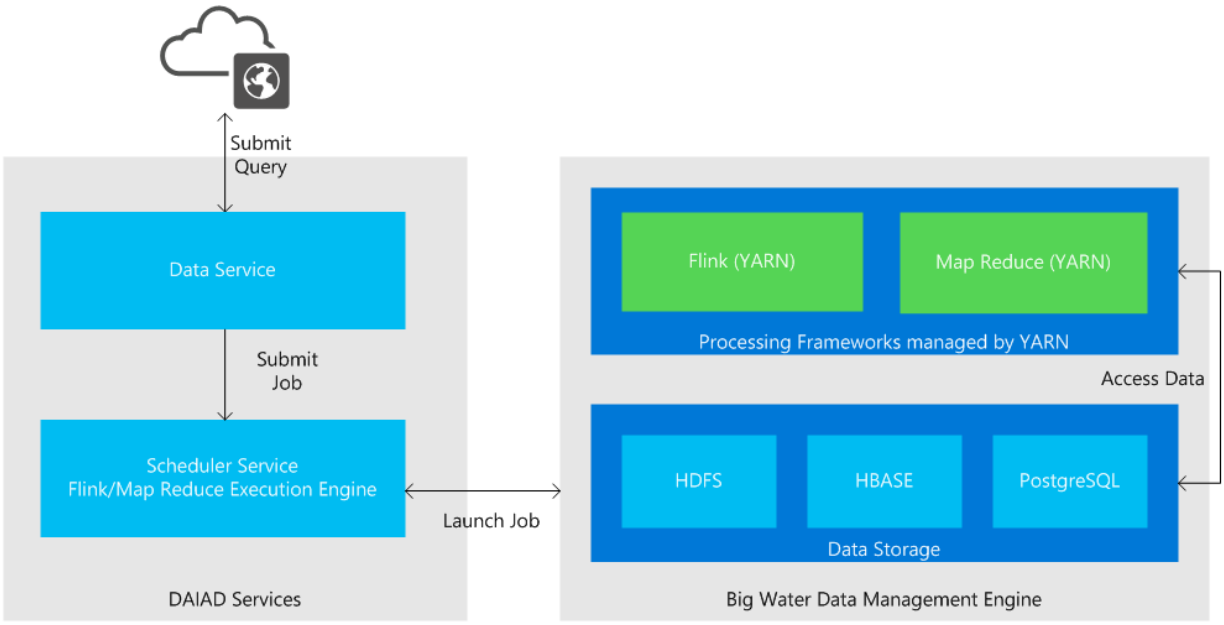


Figure 5: Job execution

1.3.1. Data API

The Data Application Programming Interface (API) supports querying data persisted by the Big Water Data Management Engine developed in WP5 and presented in Report on Prototype Deliverable D5.1.2. It is exposed as a Hypertext Transfer Protocol (HTTP) Remote Procedure Call (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 Cross-Origin Resource Sharing (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 configuration options and 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, max, count or average values. More specifically the input parameters are:

- **Timezone**: Converts the results to the specified timezone. If a timezone is not set, the timezone defined in the authenticated user's profile is used. If the user has not set her timezone, the timezone is set to UTC.
- **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. As a result, declaring a cluster is equivalent to declaring all of its groups.
    - Optionally, the users of a 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 similar to 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. When forecasting data is requested, this parameter is ignored and sum is the only value returned.
- **Source**: Declares the data source to use. When forecasting data is requested, this parameter is ignored.

Detailed documentation on the Data API syntax and request examples can be found at:

- https://app.dev.daiad.eu/docs/api/index.html.

The Data API is implemented as part of the DAIAD Services, which is presented in the Report on Prototype Deliverable 1.4. Figure 6 illustrates the Data API implementation in more detail.

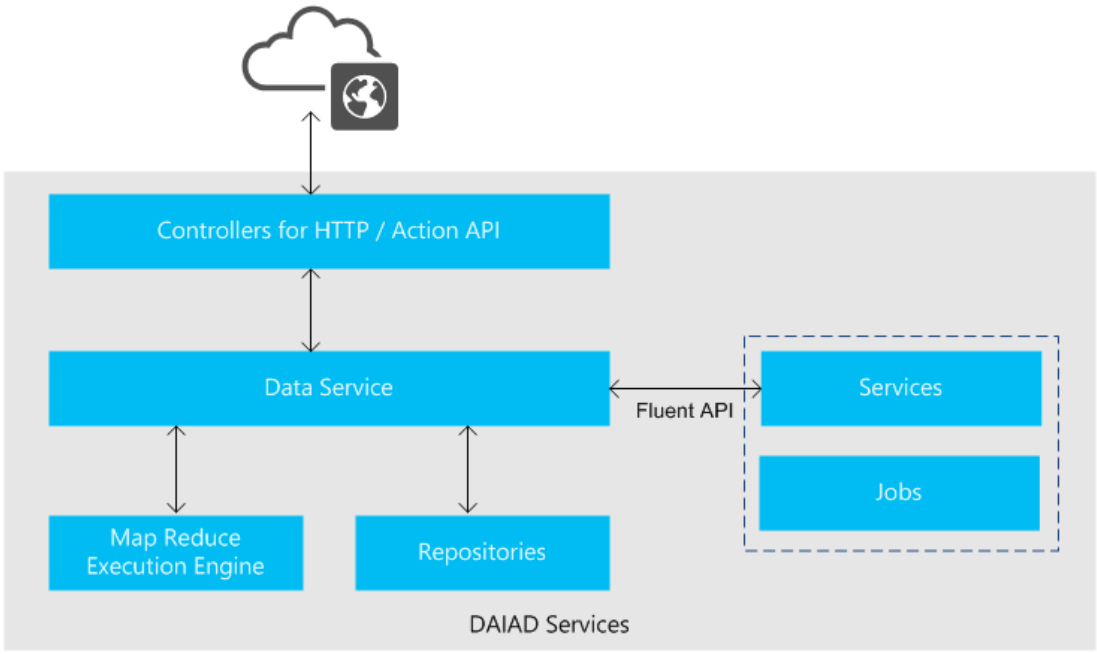


Figure 6: Data API implementation

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 submitted to the Map Reduce execution engine for computing the query results.

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[1] and the User Clustering Job[2] respectively. The former queries utility and individual user consumption data in order to generate alerts and recommendations. The latter clusters the users based on their total water consumption over a predefined time interval.

1.4. Analytics facilities

The following table summarizes the supported analytics jobs of the DAIAD system, along with a short description for each, and how it has been implemented.

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

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

Name	Category	Short Description	Implementation
User Clusters based on demographics	Clustering	Clusters uses in groups based on demographics characteristics about income, age, household size and apartment size	Java, In-process, Input/Output PostgreSQL
User Clusters based on consumption	Clustering	Clusters users in groups based on their water consumption from smart water meters for the last months	Java, MapReduce, Input HBASE, Output PostgreSQL
Smart Water Meter Data Pre-Aggregation[3]	Aggregation	Computes aggregates for daily, weekly and monthly intervals for every user over a variable time interval	Java, MapReduce, Input/Output HBASE
Data Service in-Process Implementation	Aggregation	Implements Data Service execution in-process	Java, In-process, Input HBASE
Data Service Map Reduce Implementation	Aggregation	Implements Data Service execution in-process using pre-aggregated data computed with Map Reduce	Java, In-process, Input HBASE
Households Clusters	Clustering	Finds groups of similar households based on their water consumption for the preferable time intervals	Java, Flink ML, Input/Output wrappers for HDFS
Consumers Clusters	Clustering	Finds groups of similar consumers based on their water consumption for the preferable time intervals	Java, Flink ML, Input/Output wrappers for HDFS
Households Synopsis	Aggregation	Computes total, maximum, minimum and average households' water consumption for the preferable time intervals	Java, Flink ML, Input/Output wrappers for HDFS
Consumers Synopsis	Aggregation	Computes total, maximum, minimum and mean consumers' water consumption for the preferable time intervals	Java, Flink ML, Input/Output wrappers for HDFS
Households Characteristics	Classification	Predicts households' characteristics (income, size, etc.) when contextual data (e.g., demographics,	Java, Flink, Input/Output wrappers for HDFS

[3] https://github.com/DAIAD/utility-mapreduce

		behavioral) are available based on water use	
Consumers Characteristics	Classification	Predicts consumers' characteristics (e.g., sex, age) when contextual data (e.g., demographics, behavioral) are available based on water use	Java, Flink, Input/Output wrappers for HDFS
Households Classes	Classification	Computes and predicts households' water consumption categories for arbitrary time intervals	Java, Flink, Input/Output wrappers for HDFS
Households Water Consumption	Classification - Regression	Computes and predicts households' water consumption (in liters) for arbitrary time intervals	Java, Flink, Input/Output wrappers for HDFS
Savings Potential and WaterIQ Computation	Clustering	Calculates savings potential of groups of households and their members	Java, Flink, Input/Output wrappers for HDFS

2. FML Algorithmic Framework

In this chapter, we introduce the *FML algorithmic framework* (Flink Machine Learning algorithmic framework), a collection of distributed, scalable machine learning algorithms that was implemented using Apache Flink in the context of DAIAD. These include:

- **FML-kNN**[4] **(*k-Nearest Neighbors*):** An efficient distributed k-Nearest Neighbors algorithm, able to perform either classification, or regression over *very high* volumes of water consumption data.
- **FML-SP**[5] **(*Savings Potential*):** A distributed algorithm that calculates the amount of water that can be saved by groups (and within-group) of households, derived by a scalable *k-means* algorithm.
- **FML-RF (*Random Forest*):** An efficient distributed *random forest* algorithm employed for various classification tasks.

The above algorithms are presented in the following sections.

2.1. FML-kNN

In this section, we present our work on developing FML-kNN, which has been integrated in the Consumption Analytics and Forecasting Engine to provide *predictive analytics* at the city level [CK+15]. FML-kNN supports two major machine learning processes, *classification* and *regression*, over high volumes of water consumption data. Classification enables us to categorize consumers' consumption on a city scale, assign consumers to different consumption classes according to their *behavioral* characteristics, and provide water utilities with useful insights. Regression enables us to produce scalable predictive analytics from massive amounts of water consumption data and perform forecasting tasks for the next week/month/year for the entire population. FML-kNN provides the following *analytics facilities* (please refer to the table in Section 1.4):

- Households Classes
- Households Water Consumption

In the following, we present an overview of the algorithm's components, the state-of-the-art on scalable predictive analytics, and our contributions. Next, we present some preliminaries which enable the reader to follow the models which were incorporated into our algorithms. A detailed presentation follows of the actual algorithms and their respective execution stages. The experimental evaluation of our work against competing algorithms and systems is presented in a later section.

[4] https://github.com/DAIAD/utility-flink-FML-kNN

[5] https://github.com/DAIAD/utility-flink-FML-SP

2.1.1. Overview

FML-kNN is a novel algorithm which implements the k-Nearest Neighbors *joins* method (kNN joins) [BK04]. kNN joins retrieves the nearest neighbors of *every* element in a *testing* dataset (R) from a set of elements in a *training* dataset (S) of *arbitrary dimensionality*. In the context of DAIAD, kNN joins enables us to efficiently handle *massive* amounts of water consumption and contextual data. To manage their high dimensionality, FML-kNN applies a *dimensionality reduction* method. This introduces loss of accuracy, which is partly compensated by a *data-shifting* approach. To parallelize its execution, FML-kNN partitions the data by calculating *partitioning ranges* for both testing and training datasets. As this requires costly sorting operations, it is performed on *reduced* datasets, which occur through a *sampling* approach.

The contribution of our work is the provision of two popular machine learning methods (classification and regression) that share the same distributed, Flink-based kNN joins algorithm, named FML-kNN, which is integrated in a machine learning algorithmic framework, named FML. Contrary to similar approaches, FML-kNN is executed in a *single* distributed session, despite the fact that the algorithm has *three sequential stages* which produce intermediate results. This *unified* implementation achieves better time performance and operational *efficiency* as it *eliminates* costly operations, i.e., fetching and storing the intermediate results among the execution stages.

2.1.1.1. FML-kNN components

FML-kNN provides classification and regression, two of the most important ML techniques. In the particular context of DAIAD, classification enables us to *categorize* consumers' consumption and *determine* consumption hours, while regression enables us to *forecast* the actual water consumption and *predict* consumers' characteristics.

Specifically, the components of FML-kNN and their corresponding purpose in the context of DAIAD are the following:

- F-kNN probabilistic classifier. It predicts the class/category of each element in a testing dataset (R), using historical data in a training dataset (S). Additionally, it predicts, for each household, whether consumption will occur during specific hours, or not.
- F-kNN regressor. It predicts the value of each element in a testing dataset (R), using historical data in a training dataset (S) i.e., the predicted consumption hours from the classifier.

FML-kNN is influenced by the H-zkNNJ [ZL+12] algorithm and was implemented with Apache Flink, which has various advantages compared to *Hadoop* (see Section 1.2 for details). As a result, FML-kNN inherits the advantages and characteristics of Apache Flink:

- It allows to model operations and jobs as *Directed Acyclic Graphs* (DAGs), which is more flexible than the standard MapReduce model where Map operations are strictly followed by Reduce operations.
- It allows for data *pipelining* and *in-memory data transfer* optimizations (i.e., less data preprocessing and transfer). This achieves better *efficiency* as it reduces *disk accesses* and *network traffic*.
- It efficiently supports *iterative algorithms*, which are extremely expensive in the standard MapReduce framework.

In summary, FML-kNN has three processing stages which have been unified in single Flink session and detailed in 2.1.3.3. These are:

- **Data pre-processing** (stage 1). During this stage, the dimensions of the data are *reduced* in order to minimize computational complexity.
- **Data partitioning and organization** (stage 2). During this stage, the dataset is separated into several sets, i.e. *partitions*, with each partition processed independently.
- **kNN computation** (stage 3). This stage generates the final kNNs for each data element.

2.1.1.2. Related work

Several MapReduce-based algorithms for kNN joins have been proposed in the Big Data literature. However, none of these approaches handles the specific nature of water consumption data. Song et al. [SR+15] present a review of the most efficient MapReduce-based algorithms for kNN joins and conclude that H-zkNNJ [ZL+12] algorithm outperforms in terms of completion time other similar methods. The most representative among them is *RankReduce* [SM+10] which uses *Locality Sensitive Hashing* (LSH) [IM98]. H-z*k*NNJ was developed over the *Hadoop* platform and operates in three (3) separate MapReduce sessions. Our contribution in FML-*k*NN is the *unification* of these sessions into one distributed Flink session. This achieves better time performance as it reduces multiple access and storage of data.

In the water/energy domain there are multiple applications of a kNN-based classification or regression to produce predictive analytics. Chen et al. [CD+11] use a kNN classification method to label water data and identify water usage, focusing however, on very small collections of data. Naphade et al. [NL+11] and Silipo and Winters [SW13] focused on identifying water and energy consumption patterns, providing analytics and predicting future consumptions. Similarly, they experimented on data of low granularity. Schwarz et al. [SL+12] used kNN for classification and short-term prediction in energy consumption by using smart meter data. However, they focused on an in-memory approach (*not parallel or distributed*), thus limiting the applicability of their work on larger datasets. FML-kNN partially relates to the work of Kermany et al. [KM+13], where the kNN algorithm was applied for *classification on water consumption data*, in order to detect irregular consumer behavior. Although their approach handles huge volumes of data, they were only limited on identifying irregularities and no other useful characteristics.

2.1.2. Preliminaries

In the following, we present basic concepts regarding classification and regression based on kNN joins, as well as methods for dimensionality reduction.

2.1.2.1. Classification

To classify a query dataset R (testing) containing new elements, a labelled dataset S (training) is required. A kNN joins classifier categorizes new elements via a *similarity measure* expressed by a *distance function* (e.g., Euclidean, Manhattan, Minkowski). In FML-kNN we used the *Euclidean* distance. This distance is used to obtain for each lookup element in the testing dataset R the *dominant* class (i.e., class membership) which consists of the element's k nearest neighbors. kNN classification in most cases also integrates a voting scheme, according to which, the class that appears more times among the nearest neighbors will be the resulting class. The

voting scheme can be *weighted* when someone takes into account the distances between the nearest neighbors. Then, each nearest neighbor has a weight according to its distance from the lookup element.

The set of the kNN is expressed as $X = \{x_1^{NN}, x_2^{NN}, \ldots, x_k^{NN}\}$ and the class of each one as a set $C = \{c_1^{NN}, c_2^{NN}, \ldots, c_k^{NN}\}$, the weight of each neighbor is calculated as follows [GX+11]:

$$w_i = \begin{cases} \frac{d_k^{NN} - d_i^{NN}}{d_k^{NN} - d_1^{NN}} : d_k^{NN} \neq d_1^{NN} \\ 1 \qquad\qquad\quad : d_k^{NN} = d_1^{NN} \end{cases}, \qquad i = 1, \ldots, k \qquad (1)$$

where d_1^{NN} is the closest neighbor and d_k^{NN} is the furthest one to the lookup element. By this calculation, the closest neighbors will be assigned a *greater* weight.

2.1.2.2. Regression

Regression is a *statistical* process, used to estimate the relationship between one dependent variable and one, or more, independent variables. In the machine learning domain, regression is a *supervised* method, which outputs *continuous values* (instead of discrete values such as classes, categories, labels, etc.). These values represent an estimation of the *target* (dependent) variable for the new observations. A common use of the regression analysis is the prediction of a variable's values (e.g., future water/energy consumption, product prices, web-page visibility/prediction of potential visitors), based on existing/historical data. There are numerous statistical processes that perform regression analysis, however, in the case of kNN, regression can be performed by averaging the numerical target variables of the kNN as follows.

Considering the same set of kNNs $X = \{x_1^{NN}, x_2^{NN}, \ldots, x_k^{NN}\}$ and the target variable of each one as $V = \{v_1^{NN}, v_2^{NN}, \ldots, v_k^{NN}\}$, the value of a new observation will be calculated as:

$$v_r = \frac{\sum_{i=1}^{k} v_i^{NN}}{k}, \qquad i = 1, \ldots, k \qquad (2)$$

2.1.2.3. Dimensionality reduction

The *curse of dimensionality* is an inherent challenge for data analysis tasks, which causes the increase of data *sparsity* as their dimensionality increases. In the case of kNN, it also *exponentially* increases the *complexity* of distance calculations. Further, for the particular case of DAIAD, data dimensionality increases as we combine water consumption and contextual data from the entire population. Consequently, with the volume of the data constantly increasing, a dimensionality reduction method is necessary to avoid costly distance computations.

Dimensionality reduction is a method which reduces the dimensions to one. In the literature, *space filling curves* (SFC), namely *z-order*, *Gray-code* and *Hilbert* curve (Figure 7) are often used for such a purpose. Each curve *scans* the n-dimensional space in a dissimilar manner and exhibits different characteristics w.r.t. the optimal space scanning and the computational complexity. One could argue that a space filling curve approach in the context of smart water meter consumption data could result in a similarity measure that does *not* behave like a distance in order to be leveraged by a kNN algorithm like FML-kNN. To overcome this issue, after extracting the features from the water consumption data (see 4.1.3.1), we sort them in an *ascending* manner regarding the corresponding water consumption and re-label them with an increasing number according to

their new sequence. This way, an element with a certain set of features will tend to be neighbors with elements that have similar features and, therefore, similar water consumption

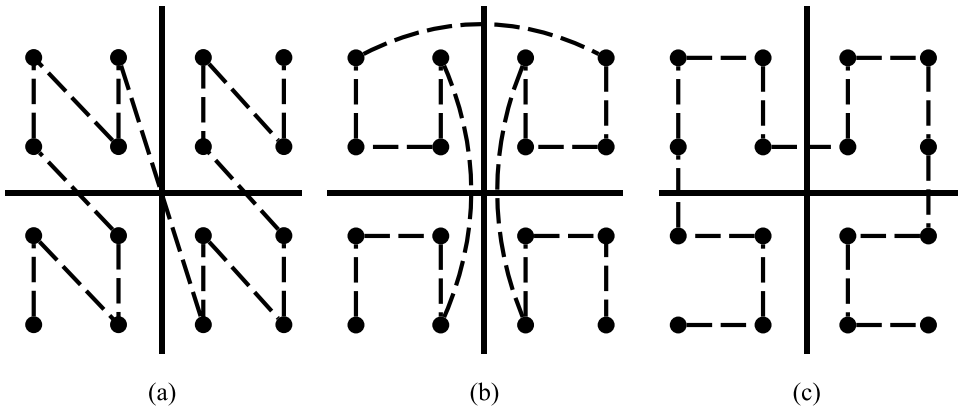


Figure 7: Space filling curves. (a) z-order curve, (b) Grey-code curve, (c) Hilbert curve

- Z-order curve (Figure 7(a)) is computed by *interleaving* the binary codes of an element's features. This process takes place starting from the most significant bit (MSB) towards the least significant (LSB). For example, the z-value of a 3-dimensional element with feature values 3 (011_2), 4 (100_2) and 5 (110_2), can be formed by first interleaving the MSB of each number (0, 1 and 1) going towards the LSB, thus forming a final value of 011101100_2. This is a *fast* process, not requiring any costly CPU execution.
- Gray-code curve (Figure 7(b)) is very similar to z-order curve as it requires only *one* extra step. After obtaining the z-value, it is transformed to Gray-code by performing *exclusive-or* operations to successive bits. For example, the Gray-code-value of 0100_2 will be calculated as follows. Initially, the MSB is left the same. Then, the second bit will be an exclusive-or of the first and second ($0_2 \oplus 1_2 = 1_2$), the third an exclusive-or of the second and third ($1_2 \oplus 0_2 = 1_2$) and the fourth an exclusive-or of the third and fourth ($0_2 \oplus 0_2 = 0_2$). Thus, the final Grey-code-value will be 0110_2.
- Finally, the computation of Hilbert curve (Figure 7(c)) requires *more complex* computations. The intuition behind Hilbert curve is that two consecutive points are nearest neighbors, thus, avoiding "jumping" to further elements, as in z-order and Gray-code curves. The curve is generated *recursively* by rotating the two bottom quadrants at each recursive step. There are several algorithms that map coordinates to Hilbert coding. We deployed the methods described in [La00] offering both forward and inverse Hilbert mapping.

2.1.3. FML-kNN

This section outlines the design and implementation of FML-kNN. The main contribution of FML-kNN is the *unification* of the three different processing stages into a single Flink session. Multi-session implementations, regardless of the distributed platform on which they are developed and operated, are significantly inefficient due to the following reasons:

- **The multiple initializations of the distributed environment.** They increase the total wall-clock time needed by an application in order to produce the final results.

- **The extra I/O operations during each stage.** They introduce latency due to I/O operations and occupy extra HDFS space.

We avoid the above issues by unifying the distributed sessions into *a single session*. Figure 8 illustrates the unified session. The stages are executed sequentially and I/O operations with HDFS take place only during the start and end of the execution.

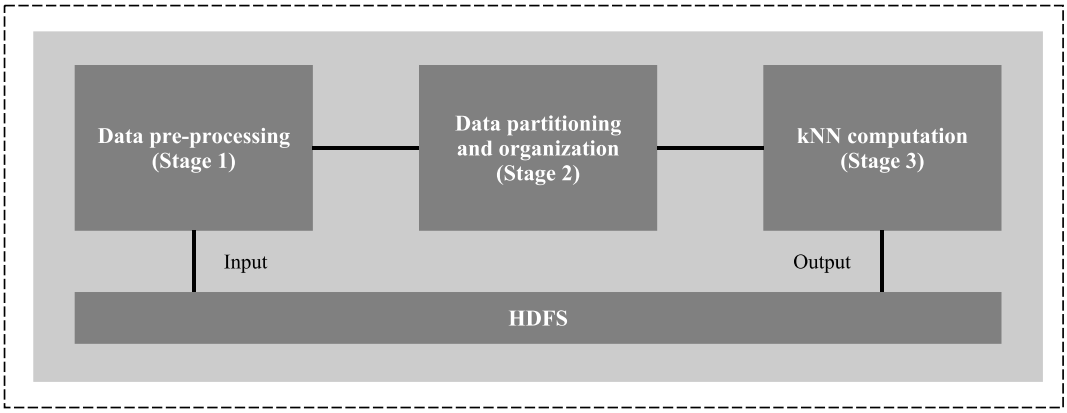


Figure 8: Single Flink session

2.1.3.1. Dimensionality reduction and shifting

To perform dimensionality reduction, we transform the elements of the input dataset into SFC values via either z-order, or Gray-code, or Hilbert curve. In our case, we enrich the consumption data with *contextual features* (temporal/seasonal/meteorological, etc.). We have 16 dimensions in our dataset which makes kNN computation very difficult especially when the elements count millions of records (see Wall-clock completion). Figure 7(a) shows the way z-order curve "fills" a two-dimensional space from the smallest z-value to the largest. We notice that some elements are *falsely* calculated being closer than others, as the curve scans them first. This in turn creates an impact on the result's *accuracy*. We overcome the latter by *shifting* all the elements by *randomly generated vectors* and repeating the process using the shifted values. This method *compensates* the lost accuracy as it enables scanning the space in an altered sequence. This is demonstrated in Figure 9.

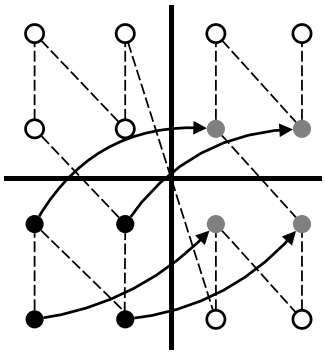


Figure 9: Data shifting. The grey dots represent the shifted values

The four bottom-left elements are shifted *twice* in the x-axis and once in the y-axis, altering the sequence in which they are scanned by the z-order curve. Consequently, taking under consideration nearest neighbors of

a point from shifted datasets, one can obtain elements that are close to it, but are missed by the un-shifted SFC. The limitation of this approach is the fact that it has to be executed *multiple times*, one for each chosen shift i $\in$ [1, α], where α is the total number of shifts.

2.1.3.2. Partitioning

A crucial part of developing a MapReduce application is the way input data are *partitioned* in order to be delivered to the required reducers. Similar baseline distributed approaches of kNN joins problem perform partitioning on both R and S datasets in $\boldsymbol{n}$ *blocks* each and cross-check for nearest neighbors among all possible pairs, thus requiring $\boldsymbol{n^2}$ reducers. This step is necessary in order to properly partition the data and feed them to the n reducers. We avoid this issue by computing $\boldsymbol{n}$ *overlapping* partitioning ranges for both R and S, using each element's SFC values. This way, we make sure that the nearest neighbors of each R partition's elements will be detected in the corresponding S partition. We calculate these ranges after properly sampling both R and S, due to the fact that this process requires costly sorting of the datasets.

2.1.3.3. FML-kNN workflow

FML-kNN has the same workflow as other similar approaches [SR+15] (see 2.1.1.2), and consists of three processing stages. The workflow of the algorithm is depicted in Figure 10. The operations that each stage of FML-kNN performs are enumerated below (the Flink operation/transformation is in parentheses):

- Data pre-processing (stage 1):
    - Performs dimensionality reduction via SFCs on both R and S datasets (**Flat Map R/S**).
    - Shifts the datasets (**Flat Map R/S**).
    - Unifies the datasets and forwards to the next stage (**Union**).
    - Samples the unified dataset (**Flat Map Sampling**).
    - Calculates the partitioning ranges and broadcasts them to the next stage (**Group Reduce**).
- Data partitioning and organization (stage 2):
    - Partitions the unified dataset into n partitions, using the received partitioning ranges (**Flat Map**).
    - For each partition and each shifted dataset, the kNNs of each element in dataset R are calculated (**Group Reduce**).
- kNN computation (stage 3):
    - The final kNNs for each element in dataset R are calculated and classification or regression is performed (**Group Reduce**).

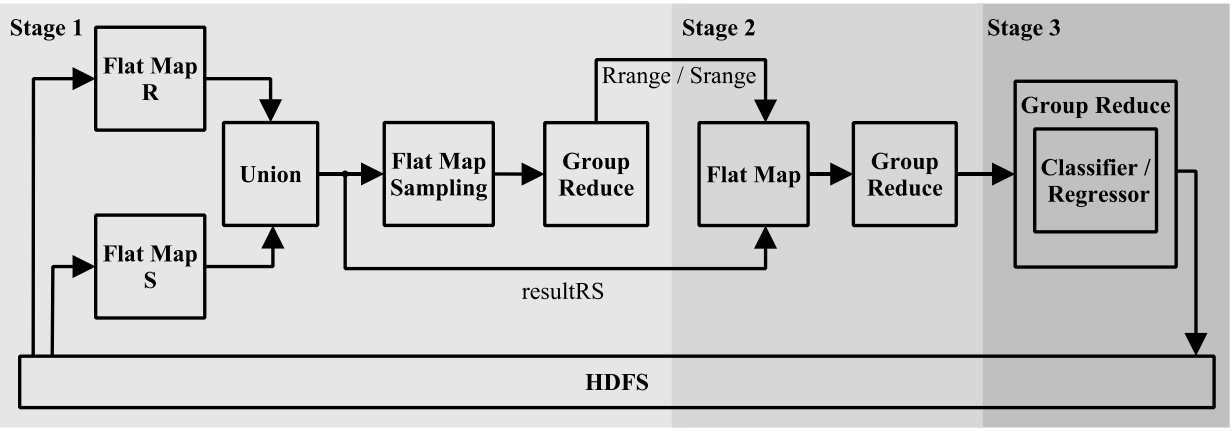


Figure 10: Single session FML-kNN

In the following, we present each stage in more details.

2.1.3.3.1. Data pre-processing (stage 1)

The R and S datasets are read as *plain text* from HDFS and delivered to two separate concurrent `flatMap` transformations, identifiable by the input source file. During this stage, the SFC values (z-values for z-order, h-values for Hilbert and g-values for Grey-code curve) and possible shifts, are calculated and passed to a `union` transformation, which creates a *union* of the two datasets. The unified and transformed datasets are then forwarded to the next stage (stage 2) and to a *sampling process*, which is performed by a separate `flatMap` transformation. The sampled dataset is then passed on a `groupReduce` transformation, grouped by a shift number. This way, α (number of shifts) reducers will be assigned with the task of calculating the partitioning ranges for R and S datasets, which are then *broadcast* to the next stage (data partitioning and organization). The left part of Figure 10 depicts this process.

Algorithm 1 presents the pseudocode of stage 1. The random vectors are initially generated and cached on HDFS in order to be accessible by all the nodes taking part in the execution. They are then given as input to the algorithm, along with datasets R and S. Notice that $v_1 = \vec{0}$, which indicates that the datasets are not shifted during this iteration. This process takes place α times, where α is the number of shifts (Line 5). After shifting the datasets, during the first mapping phase (Lines 5-9), the elements' SFC values are calculated and collected to R_i^T and S_i^T $(i = 1, \ldots, \alpha)$. Then, the sampling is performed by the second mapping phase (Lines 10-17). During the reduce phase (Lines 18-22), the partition ranges ($\mathbf{Rrange_i}$ and $\mathbf{Srange_i}$) for each shift are calculated using the sampled datasets and broadcast to stage 2 (Lines 18-20). The output is the unified transformed datasets, which finally feed the *data partitioning and organization stage* (Line 22).

```
1  ▷ The pre-processing stage's input
   Input: Datasets R, S and random vectors V = {v_1, ..., v_α}, v_1 = 0⃗
2  ▷ The output, that will be emitted to the next stage
   Output: Transformed datasets R_i^T and S_i^T, i = 1, ..., α
3  begin
4      ▷ The same procedure is repeated for each shift
5      for i = 1, ..., α do
6          R_i = R + v_i
7          S_i = S + v_i
8          R_i^T ← CALCSFC(R_i)
9          S_i^T ← CALCSFC(S_i)
10         foreach x ∈ R_i^T ∪ S_i^T do
11             ▷ Sampling
12             r ← RANDOM(0, 1)
13             if r < MinThreshold then
14                 if x ∈ R_i then
15                     INSERTSAMPLE(s, R̂_i^T)
16                 else if x ∈ S_i then
17                     INSERTSAMPLE(s, Ŝ_i^T)
18         Rrange_i ← CALCRANGE(R̂_i^T)
19         Srange_i ← CALCRANGE(Ŝ_i^T)
20         BROADCAST(Rrange_i, Srange_i)
21         ▷ Emit to the partitioning and organization stage
22         return R_i^T ∪ S_i^T
```

Algorithm 1: Data pre-processing (stage 1)

2.1.3.3.2. Data partitioning and organization (stage 2)

The transformed datasets of the stage 1 are partitioned to $n \times \alpha$ blocks via a custom partitioner, after fetching the previously broadcast partitioning ranges. Each block is then delivered to a *different* reducer through a `groupBy` operation. Finally, the nearest neighbors for each lookup element are calculated via proper range search operations and passed on the next stage (kNN computation). The middle part of Figure 10 depicts this process.

Algorithm 2 presents the pseudocode of stage 2. During the map phase (Lines 7-18), after having read each shift's broadcast partition ranges (Line 6), the received transformed datasets are *partitioned* into $\alpha \times n$ buckets ($R^{g\,x\,i}$ and $S^{g\,x\,i}$, $i = 1, \ldots, \alpha,\ g = 1, \ldots, n$) Lines 13 & 18), α being the number of shifts and *n* the number of partitions. The partitions $S^{g\,x\,i}$ are then sorted and emitted to the reducers (Lines 23-30) along with the corresponding partitions $R^{g\,x\,i}$. There, for each *x* ∈ R element, a range search is performed on the proper sorted partition in order for its kNN to be determined (Line 23) and its initial coordinates are calculated (Line 24). The initial coordinates of all neighboring elements' coordinates are then calculated (Line 26) and their distance to the *x* ∈ R element is computed (Line 27). Finally, all nearest neighbors are integrated into the proper dataset ($R_{k\,x\,\alpha}$, Line 28) along with the calculated distance and feed the stage 3 (*kNN computation*), grouped by *x* ∈ R elements.

```
1  ▷ This stage's input are the datasets emitted during the partitioning and organization stage
   Input: Datasets R_i^T, S_i^T, i = 1, ..., α
2  ▷ The output, that will be emitted to the next stage
   Output: Dataset R_{k×α}
3  begin
4      ▷ Initialization of the dataset to be emitted
5      R_{k×α} = ∅
6      RECEIVEBROADCAST(Rrange_i, Srange_i)
7      ▷ Again, repeat for each shift
8      for i = 1, ..., α do
9          ▷ Partition the R elements
10         foreach x ∈ R_i^T do
11             for g = 1, ..., n do
12                 if ZVAL(s) ∈ Rrange_i(g) then
13                     ADDINTOPARTITION(s, R^{g×i})
14         ▷ Partition the S elements
15         foreach x ∈ S_i^T do
16             for g = 1, ..., n do
17                 if ZVAL(s) ∈ Srange_i(g) then
18                     ADDINTOPARTITION(s, S^{g×i})
19         for g = 1, ..., n do
20             ▷ Sorting is needed to properly perform range search
21             SORT(S^{g×i})
22             foreach x ∈ R^{g×i} do
23                 RES ← RANGESEARCH(x, k, S^{g×i})
24                 CC_x ← CALCCOORDS(x)
25                 foreach neighbour ∈ RES do
26                     CC_neighbour ← CALCCOORDS(neighbour)
27                     CD ← CALCDIST(CC_x, CC_neighbour)
28                     R_{k×α} ← ADD(x, neighbour, CD)
29     ▷ Emit to the final stage, grouped by element
30     return R_{k×α}
```

Algorithm 2: Data partitioning and organization (stage 2)

2.1.3.3.3. kNN computation (stage 3)

The calculated α x k NNs of each R element, are received from stage 2 and forwarded to |R| reduce tasks, which calculate the *final* kNNs and perform *either* classification or regression, depending on *user preference*.

We extend this approach by assigning a probability to each of the candidate classes for the lookup element. We consider the set $P = \{p_j\}_{j=1}^{l}$ where I is the number of classes. The final probability for each class is derived as follows:

$$p_j = \frac{\sum_{i=1}^{k} w_i \cdot I(c_j = c_i^{NN})}{\sum_{i=1}^{k} w_i}, \qquad j = 1, \ldots, l \qquad (3)$$

where $I(c_j = c_i^{NN})$ is a function which takes the value 1 if the class of the neighbor x_i^{NN} is equal to c_j.

Finally, we classify the element as:

$$c_r = \underset{c_j}{argmax}\, P, \qquad j = 1, \ldots, l \qquad (4)$$

which is the class with the *highest probability*. This method is used for each element in the testing dataset (R).

In this stage, we calculate either the probability for each class (classifier) or the final value (regressor) for each element in R. The result is stored on HDFS in *plain text*. More details on the several data formats in the course of the algorithm's execution can be found in Annex: Evaluation datasets. The right part of Figure 10 depicts this process.

Algorithm 3 presents the pseudocode of stage 3. During the stage 3 of the algorithm, which consists of *only a reduce operation*, kNNs of each R element are fetched from the grouped set of $R_{k\,x\,\alpha}$. Finally, for each lookup element either classification (Line 9) or regression (Line 11) is performed, after determining its final nearest neighbors (Line 7). The results are added to the resulting dataset (Line 9), which is then stored on HDFS (Line 12) in the proper format.

```
1  ▷ The input is the grouped by element dataset emitted during the previous stage
   Input: Datasets R, R_{k×α}
2  ▷ The algorithm's results
   Output: Dataset R_f
3  begin
4      ▷ Initialization of the final dataset
5      R_f = ∅
6      foreach x ∈ R do
7          RES ← kNN(x, R_{k×i})
8          if classification then
9              FIN ← CLASSIFY(RES)
10         else if regression then
11             FIN ← REGRESS(RES)
12         R_f ← ADD(s, FIN)
13     ▷ Store the final results on HDFS
14     return R_f
```

Algorithm 3: kNN computation (stage 3)

2.2. FML-SP

In this section, we present FML-SP, our savings potential algorithm, also integrated in the Consumption Analytics and Forecasting Engine. It operates over the SWM dataset (see Annex: Evaluation datasets) and its purpose is to provide the following water consumption related insights:

- Savings Potential: The amount of water (in liters or in percentage) that a household, or a group of households can save during a specific period. It is provided both in liters and in percentage per group, compared to the rest of the groups.
- WaterIQ Score: A rating ranging from A to F, assigned to each household within each group, that describes the water saving behavior of the household, compared to the rest of the households within the same group.

The above insights provide the *Savings Potential and WaterIQ Computation* analytics facility (please refer to the table in Analytics facilities).

In the following, we present an overview of the algorithm, followed by a description of the scalable k-means clustering algorithm that is its main component. Finally, we outline FML-SP, the top-level algorithm that computes the final savings potential and Water IQ score.

2.2.1. Overview

FML-SP is consisted of the following two components, both implemented in order to execute on a *distributed environment*, using Apache Flink:

- A *scalable* k-means *clustering* algorithm.
- A savings potential and WaterIQ score computation algorithm.

FML-SP's main component implements clustering, the third *major* machine learning processes (apart from classification and regression), which can operate over *high volumes* of water consumption data. Clustering enables us to cluster the households from the SWM dataset into groups according to their water consumption behavior. This is achieved by computing their *average water consumption weekly time series* for a pre-specified time interval (e.g., during a month) and then perform clustering on the resulting time series, to obtain a pre-determined number of groups of households. The clustering method that we chose was *k-means*, for reasons that will be described in the following sections.

The resulting groups enable us to calculate the savings potential per group and the *within-group* Water IQ score of each group's members. The savings potential for each group is the total water volume above the group's *average* per-household consumption during the selected period. It is provided both in liters and in percentage, compared to the rest of the household groups. Finally, the Water IQ score is obtained by generating consumption *buckets* within each group for the selected period and placing each household in a certain bucket according to their total water consumption during the same period.

2.2.2. Scalable k-means Clustering

Towards efficiently performing clustering of the users according to their water consumption time series, we developed a Flink-based k-means [HM79] clustering algorithm, integrated in our Consumption Analytics and Forecasting Engine and provides the following analytics facilities (please refer to the table in Analytics facilities):

- Households Clusters
- Consumers Clusters

The algorithm extends an existing Apache Flink-based k-means algorithm and is currently being integrated in the Apache Flink *machine learning library*[6].

2.2.2.1. Motivation

There are several motivating factors for choosing k-means as our default clustering algorithm. First, k-means, as a *centroid-based* method, has a significantly *lower complexity* than other clustering methods, which makes it ideal for clustering Big Data, such as the data we receive in DAIAD. As several evaluating works [SS12, PS+12,

[6] http://bit.ly/2qM8fVr

Ab08, SG14] in the literature suggest, k-means achieves the best performance in terms of *execution time* and *scalability* (performs better on very large datasets), which in our case is crucial, due to the very large size of our datasets and the need for the results to be available in near real-time. Furthermore, k-means produces more accurate, *tighter* clustering results than similar methods [JK+14], which makes it one of the most efficient clustering algorithms and an ideal choice in our case.

2.2.2.2. Implementation

We extended[7] the available k-means algorithm of FlinkML[8] by *augmenting* its applicability towards efficient clustering on data from various sources. More specifically, we added the option of clustering time series data, by *raising* its input dimensionality capabilities to a *user-specified* number of dimensions. Furthermore, we provided the option of choosing between either *Euclidean*, or *Dynamic Time Warping* [DC94] as a distance measure. The latter allows for efficient matching of similar time series that might have differences in terms of scale, or in terms of horizontal shifts in the time axis. Figure 11 depicts a high-level flow diagram of the algorithm.

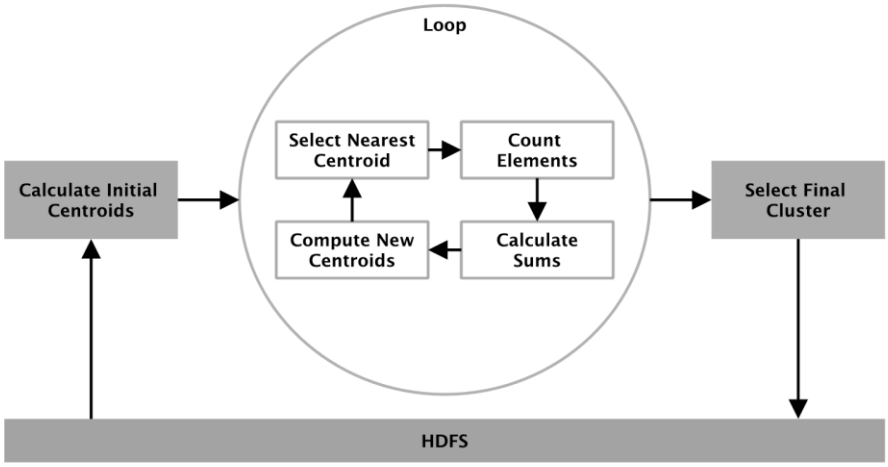


Figure 11: High-level overview of the implemented k-means algorithm

Initially, the dataset in the form if time series is read from HDFS and a reduce operation calculates the initial *centroids* of the algorithm (*Calculate Initial Centroids*). The results are passed onto the main loop of the algorithm (*Loop*), which iterates a *user-defined* number of times. During each loop, the nearest centroid for each element is selected via a reduce operation (*Select Nearest Centroid*), followed by a map operation that appends a count indicator for each element of each cluster (*Count Elements*). A following reduce operation counts the elements of each cluster and finally, the new centroids are calculated using a map operation (*Compute New Centroids*). When the maximum number of iterations is reached, the loop terminates and a final reduce operation assigns each element to its final cluster (*Select Final Cluster*).

[7] https://github.com/DAIAD/utility-flink-FML-RF

[8] https://github.com/apache/flink/blob/master/flink-examples/flink-examples-batch/src/main/java/org/apache/flink/examples/java/clustering/KMeans.java

2.2.3. Savings Potential and WaterIQ Score

The savings potential of a household is an estimate of the household's *inelastic* consumption, i.e., the absolute minimum consumption required to sustain safe and healthy well-being. This estimate is important as it can be used to calculate the savings potential of a population group (e.g., neighborhood, large families), applied to deliver *personalized* demand goals/limits, a fair resource budget per household in cases of urgent restrictions, as well as novel pricing schemes.

Our approach employs k-means clustering to generate clusters of similar households, and estimates the savings potential by comparing the household's consumption against the cluster it belongs to. Figure 12 illustrates the flow of distributed operations that are required to calculate the savings potential and WaterIQ score for a set of customers. Initially, the dataset in the form of hourly consumption is read from HDFS and a reduce operation calculates the *per-customer* average time series of a given granularity (i.e., daily, weekly, monthly) for a pre-determined period (e.g., January). The latter are forwarded to the k-means clustering algorithm, which groups the customers in a number of clusters, determined by a *rule of thumb* described by Kodinariya et al. in [KP13], which calculates k as follows:

$$k \approx \sqrt{n/2} \quad (5)$$

where n is the number of elements to be clustered. The reason for employing this rule of thumb is the fact that it provides a *very fast* estimation of the optimal number of clusters, while other approaches would significantly slow down the process. A following reduce operation calculates the WaterIQ score per customer and per cluster, by placing each one in a certain total consumption bucket during that period. The buckets are calculated by obtaining the maximum and minimum total consumption per cluster and dividing the resulting interval into equally sized buckets. This way we avoid slowing down the process, maintaining its time efficiency in high levels. Nevertheless, the users (i.e., water utilities) have the option to provide their own bucket sizes and number of clusters.

Example 1: After clustering the households according to their average weekly time series during March, we obtain two clusters: "Cluster A" and "Cluster B" with 3 members each. "Household 1" from "Cluster A" spent 3,000 liters during March, "Household 2" 5,000 liters and "Household 3", 9,000 liters. Thus, the resulting buckets and corresponding WaterIQ scores for "Cluster A" are:

- Bucket 1 (A): $cons < 4000\ liters$
- Bucket 2 (B): $4000\ liters \leq cons < 5000\ liters$
- Bucket 3 (C): $5000\ liters \leq cons < 6000\ liters$
- Bucket 4 (D): $6000\ liters \leq cons < 7000\ liters$
- Bucket 5 (E): $7000\ liters \leq cons < 8000\ liters$
- Bucket 6 (F): $cons \geq 8000\ liters$

Thus, according to the above, "Household 1" will be assigned a WaterIQ score of A, "Household 2" a C and "Household 3" an F. The WaterIQ scores for "Cluster B" are calculated similarly.

Finally, the savings potential per cluster for the predetermined period, is calculated through a final reduce operation, by summing the total consumption of the households that were above the cluster's average total consumption.

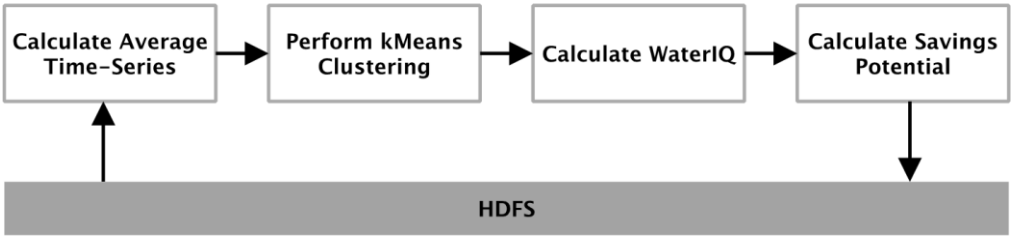


Figure 12: Savings potential

Example 2: Consider the same clusters, "Cluster A" and "Cluster B" from Example 1 above, that resulted after performing k-means clustering. As previously, Household 1" from "Cluster A" spent 3,000 liters during March, "Household 2" 5,000 liters and "Household 3", 9,000 liters. For "Cluster B", "Household 4" spent 2,000 liters, "Household 5" 10,000 liters and "Household 6" 14,000 liters. Thus, the average per-household water consumption during March for "Cluster A" was:

$$\frac{3000 + 5000 + 9000}{3} \approx 5667\ liters$$

and for "Cluster B":

$$\frac{3000 + 5000 + 9000}{3} \approx 8667\ liters$$

Consequently, the savings potential for each household of each cluster is the following:

- Cluster A
  - Household 1: $3000 - 5667 = -2667$, which is negative, so 0 liters.
  - Household 2: $5000 - 5667 = -667$, which is negative, so 0 liters.
  - Household 3: $9000 - 5667 = 3333$, which is positive, so 3,333 liters.
- Cluster B
  - Household 4: $2000 - 8667 = -6667$, which is negative, so 0 liters.
  - Household 5: $10000 - 8667 = 1333$, which is positive, so 1,333 liters.
  - Household 6: $14000 - 8667 = 5333$, which is positive, so 5,333 liters.

The total savings potential of the population is 3,333 + 1,333 + 5,333 = 9,999 liters and, for each cluster, the savings potential in liters and in percentage (in parentheses), compared to each other is:

- **Cluster A:** 3,333 liters (≈33%).
- **Cluster B:** 6,666 liters (≈67%).

2.3. FML-RF

In this section, we present FML-RF, a *random forest* [Ti95] classification algorithm, which is part of our FML algorithmic framework and operates over very large amounts of data of *various* nature collected from arbitrary sources, either in the form of time series or *cross-sectional* data. Its purpose is the efficient classification of very large amounts of such data and its many advantages (see Overview) over similar methods make it an attractive choice for many *data mining* applications. In the context of DAIAD, it supports *scalable demographics prediction* and extracting useful information regarding important *water demand determinants*. The algorithm, during building the random forest shares functionality with FML-kNN (see FML-kNN) in its *partitioning phase*. FML-RF provides the following analytics facilities (please refer to the table in Analytics facilities):

- Consumers Characteristics
- Households Characteristics

In the following, we present an overview of the algorithm, including our motivation and the current state-of-the-art in distributed random forest approaches. Finally, we describe FML-RF's architecture and its implementation.

2.3.1. Overview

As mentioned, FML-RF is a distributed random forest method, developed with Apache Flink, similarly to the rest of the FML framework's algorithms. FML-RF, can *simultaneously* classify a very large batch of input elements. Similarly to kNN joins, it receives a training dataset (S) and a testing dataset as input and classifies all the elements in the testing dataset, using the ones in the training dataset. It is consisted of two main components:

- **Building:** The random forest is built, using the training dataset (S).
- **Querying:** The elements of the testing dataset (R) are classified using the produced random forest.

As mentioned, during building, FML-RF shares functionality with FML-kNN in its partitioning phase, where the input datasets are undergone dimensionality reduction via space filling curves and are distributed among the reducers of the application. The reducers then create a *local random forest* with the partition they receive. The various random forests are finally merged per partition into a unified one, which is stored on HDFS. To our knowledge, distributed random forests did *not* receive much attention by the research community. A state-of-the-art approach is [LC+12] where Li et al. make use of histograms to calculate the best split conditions. However, the calculation of the histograms can be expensive in high dimensional datasets, thus, reducing scalability.

2.3.2. Motivation

The random forest is a popular machine learning classification method, that owes its popularity in the following advantages:

- **Accuracy**: Random forest is one of the most *accurate* learning algorithms.

- **Efficiency**: It runs efficiently on *large* amounts of data, even in a centralized form. A distributed implementation will *multiply its efficiency*.
- **High dimensionality applicability**: It can operate on a *large* number of input variables.
- **Feature importance extraction**: It has the ability to *automatically generate* and output its input features' *importance*, which allows us to extract the resource consumption determinants' *sensitivity*.
- **Missing/erroneous data robustness**: It can maintain its accuracy in high levels, even when a large proportion of the data are *missing* or are *erroneous*.
- **Unbalanced dataset robustness**: It maintains high accuracy on *unbalanced* datasets, i.e., datasets whose target variable classes are not equally distributed.
- **Outlier detection**: It computes *proximities* between pairs of cases towards locating outliers.
- **Clustering applicability**: It can be extended to operate on *unlabeled* data, leading to *unsupervised* clustering and outlier detection.

We performed thorough benchmarking regarding the prediction of household characteristics, during which we tested the following state-of-the-art machine learning classification methods:

- *Support Vector Machines* (SVM)
- *Neural Networks*
- *Decision Trees*
- *Logistic Regression*

The procedure took place as follows. Initially, we extracted the following features from the trial users' smart water meter and demographics data:

- Consumption-related
    - Mean household consumption
    - Average per-day zero-consumption hours
    - Average maximum hourly consumption per day
- Demographics-related
    - Number of household members
    - Number of showerheads in the household
    - Apartment size
    - Number of male residents
    - Number of female residents
    - Number of children residents
    - Household's total income

- An indicator whether they own the house or rent it

We extracted the above features separately for *weekdays* (i.e., days that people work) and *holidays* (i.e., weekends or local holidays in Alicante) and measured the *correlation* between the consumption and demographics-related ones. We noticed that the correlations were adequately high to attempt a prediction of the latter, using the former. Thus, we generated our final dataset, which was consisted of *six* features: the consumption-related features for the weekdays and the consumption-related features for the holidays. We finally applied all the above algorithms, obtaining the following results:

Table 1: Machine learning algorithms comparison for demographics prediction

Household Characteristic	Algorithm			
	Random Forest	SVM	Logistic Regression	Neural Network
Apartment size	79%	81%	60%	69%
Household members	72%	68%	47%	62%
Number of children	75%	70%	47%	52%
Number of males	74%	74%	51%	59%
Number of females	76%	70%	50%	53%
Income	60%	61%	34%	30%

As presented in Table 1, the random forest performed better than all the rest of the algorithms, in predicting all household characteristics (except the apartment size), with the SVM performing slightly worse. These results, combined with the aforementioned advantages of the random forest algorithm were the main reasons for deciding to develop a distributed implementation of it. Finally, apart from scalable and accurate demographics prediction, FML-RF's high robustness on missing/erroneous data will be exploited in our future work to address more effectively the *low veracity* of water consumption data, a critical issue hindering real-world smart water metering infrastructures not gaining adequate attention from the research community.

2.3.3. Scalable Random Forest

The building component of FML-RF is consisted of the following three main distributed stages:

- **Partitioning (Stage 1 - Map)**: During this operation, the input data are read from HDFS, the z-values are calculated and the training dataset is partitioned via a sampling method, similarly to the stage 1 of FML-kNN, before pushed on the next stage.
- **Decision Trees Computation (Stage 2 - Reduce)**: A pre-determined (given as input) number of random decision trees for each reducer are calculated using the received partitions of the training dataset. They are essentially pushed to the next stage.

- **Merging (Stage 3 - Reduce)**: The received decision trees are horizontally merged to form the final random forest, which is then stored on HDFS.

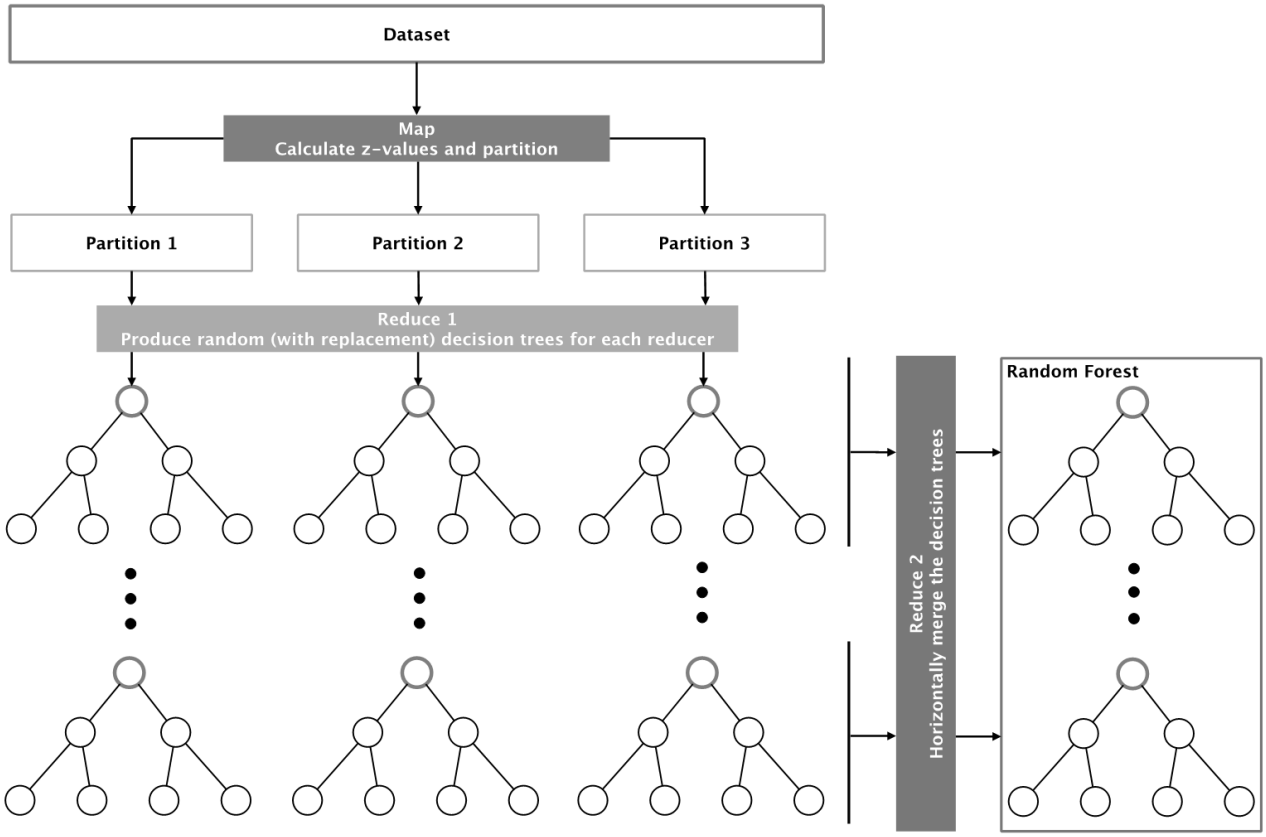


Figure 13: The Distributed Random Forest algorithm

Figure 13 depicts the structure of FML-RF. Initially, the dataset is read from HDFS and dimensionality reduction is performed, via either z-order, Grey-code, or Hilbert space filling curves. Then, the dataset is *sorted* according to the SFC value and partitioned by a map operation. The resulting partitions are distributed among several reducers, each of which generates a predefined number of random (with *replacement*) decision trees, that form a *local* random forest on that partition. A second reduce task horizontally dispenses the local random forests to a second group of reducers, grouped by the decision tree number. Each reducer *merges* the decision trees it receives and stores the resulting one on a specific path on HDFS. Ultimately, all the stored trees comprise the final random forest.

3. The BTSR-Tree Hybrid Index

In this section, we present our work on developing the *BTSR-Tree hybrid index* [CS+17], which indexes *geolocated* time series (i.e., time series annotated with location) and is able to efficiently answer various *hybrid* queries (i.e., queries that involve geolocated time series) by performing effective pruning on both time series and spatial domain. In the following, we present an overview followed by the state-of-the-art on time series and hybrid indexing. Next, we present some preliminaries which enable the reader to comprehend the theoretical foundations of our work. A detailed presentation of the actual algorithm follows. The experimental evaluation of our work demonstrates a speed-up of 1.5 to 5 times in hybrid query workloads.

3.1. Overview

Time series associated with specific locations, such as visitor check-ins or sensor readings, have increased in size and popularity in several domains. Although several works have focused on efficient time series similarity search, there has been limited attention to the inherent challenge that geolocated time series introduce for hybrid similarity queries on both spatial proximity and time series similarity. In the context of DAIAD, all the time series that are collected (i.e., SWM and shower) are geolocated, since the exact location of each household is already known. We, thus, enable the computation of queries such as "*retrieve all the households that are located close to the given coordinates and consume water in a similar manner to the given time series*".

We have implemented a hybrid index for efficiently supporting *similarity search* on geolocated time series, combining both spatial proximity and time series similarity. The index, called BTSR-tree, is an extension of the *R-tree* spatial index. In the BTSR-tree, each node is augmented with additional information corresponding to the bounds of the time series contained in its subtree, in addition to the standard *Minimum Bounding Rectangle* (MBR) denoting the spatial bound of its contents. Maintaining both kinds of bounds in each node allows to prune the search space *simultaneously* in the spatial dimension and in the time series dimension while traversing the index.

The contribution of our work includes the solution of the problem of similarity search for geolocated time series, via hybrid *boolean* or *top-k* queries combining both *spatial proximity* and *time series similarity*. This is achieved by the BTSR-tree, a hybrid index for geolocated time series, extending the spatial R-tree and augmenting each node with appropriate time series bounds. We experimentally validate our proposed approach using real-world datasets from different application use cases (please refer to Experimental Evaluation), showing that our hybrid indices can *effectively* allow *simultaneous pruning* of the search space in both spatial and time series domains, significantly *reducing* the required number of *node accesses*.

3.1.1. Related Work

In this section, we first present an overview of related work on time series indexing and then we discuss hybrid spatio-textual indices.

3.1.1.1. Time Series Indexing

Earlier approaches towards indexing time series data were based on leveraging multi-resolution representations. For instance, the *Discrete Wavelet Transform* [Gr95] is used in [CF99] to gradually reduce the dimensionality of time series data via the *Haar wavelet* [Ha10] and generate an index using the coefficients of the transformed sequences. In [PM02], it is further observed that, other than orthonormal wavelets, bi-orthonormal ones can also be used for efficient similarity search over wavelet-indexed time series data, demonstrating several such wavelets that outperform the Haar wavelet in terms of precision and performance. In addition, an alternative approach to the k-nearest neighbour search over time series data is introduced in [KK11]. The proposed method accesses the coefficients of Haar-wavelet-transformed time series through a sequential scan over step-wise increasing resolutions.

State-of-the-art approaches for time series indexing comprise methods based on the *Symbolic Aggregate Approximation* (SAX) representation [LK+07]. This is derived from the *Piecewise Aggregate Approximation* (PAA) representation of a time series [KC+01, YF00], by quantizing the segments of its PAA representation on the y-axis. The first attempt to leverage the potential of the SAX representation was presented in [SK08], introducing the *indexable* Symbolic Aggregate Approximation (iSAX), capable of a multi-resolution representation for time series. The iSAX index was further extended to iSAX 2.0 in [CP+10] by enabling bulk loading of time series data. Its next version is the iSAX2+ index [CS+14], which handles better the expensive I/O operations caused by the aggressive node splitting while building the index. Finally, the ADS+ index [ZI+14] is another extension of iSAX, which attempts to overcome the still significantly expensive index build time by *adaptively* building the index while processing the workload of queries issued by the user. A comprehensive overview and comparison of the time series indexing approaches based on the SAX representation is presented in [Pa16]. However, *none* of the above approaches supports geolocated time series, and thus cannot efficiently process hybrid queries combining time series similarity with spatial proximity.

3.1.1.2. Hybrid Indices

The *vast majority* of works towards hybrid indexing focus on spatio-textual objects, e.g., *Points of Interest* (PoI) with textual descriptions, such as *geotagged tweets* or *posts in social media*, etc. This has motivated research on hybrid spatial-keyword queries combining location-based predicates with keyword search. Main query types include the *Boolean Range Query*, which retrieves all objects that contain a given set of keywords and are located within a specified spatial range; the *Boolean kNN Query*, which returns the k nearest objects to a specific location and contain the given keywords; and the *Top-k kNN Query*, which finds the top-k objects according to an objective function that assigns hybrid scores to objects based on both their keyword similarity and spatial proximity to the query object [CC+13].

To evaluate such queries efficiently, the main idea is to construct hybrid index structures that simultaneously partition the data in both dimensions, spatial and textual. Essentially, this implies combining a spatial index structure (e.g., R-tree, *Quadtree*, *Space-Filling Curve*) with a textual index (e.g., *inverted file*, *signature file*). Depending on their form, the resulting variants can be characterized either as *spatial-first* or *textual-first* indices [CH+11]. One of the most fundamental and characteristic ones is the IR-tree [CJ+09, LL+11], which extends the R-tree by augmenting the contents of each node with a pointer to an inverted le indexing terms and documents contained in its sub-tree. Several other hybrid spatio-textual indices extending the R-tree (or R -

tree) have been proposed, such as the IR-tree [FH+08], the KR-Tree [HH+07], SKI [CW+10] and S2I [RG+11], while methods based on space filling curves include SF2I [CS+06] and SFC-QUAD [CH+11].

All the aforementioned approaches focus exclusively on combining spatial queries with keyword search. To the best of our knowledge, our work is the *first* one to address geolocated time series, combining spatial queries with similarity search for time series.

3.2. Preliminaries

In the following, we present basic concepts regarding geolocated time series and methods that apply similarity search on them. We also introduce and formulate the hybrid queries that the index can calculate.

3.2.1. Similarity Search for Geolocated Time Series

A time series is a time-ordered sequence of values $T = \{v_1, \dots, v_w\}$ where v_i is the value at the i-th time point and w is the length of the series. In our work, each time series is also characterized by a *location*, denoted by $T.loc$. For brevity, in the sequel such geolocated time series will be also called objects. Assuming a 2-dimensional space, we further use the notation $T.loc_x$, $T.loc_y$ to refer to the (x, y) coordinates of T 's location. Next, we define the distance measures we employ for similarity search on geolocated time series.

- **Spatial Distance**: The distance between two geolocated time series T and T' is calculated using the Euclidean distance of their respective locations. Furthermore, we normalize this distance with $maxDist_{sp}$, i.e., the maximum spatial distance of any pair of objects in the dataset, to obtain a measure in the interval $[0,1]$. Thus:

$$dist_{sp}(T, T') = \frac{\sqrt{(T.loc_x - T'.loc_x)^2 + \left(T.loc_y - T'.loc_y\right)^2}}{maxDist_{sp}} \qquad (6)$$

- **Time Series Distance**: In the time series domain, similar to many prior works (e.g., [SK08]), we also apply the Euclidean distance to measure the similarity of a pair of objects. Specifically, we calculate the distance between two time series T and T' as follows:

$$dist_{ts}(T, T') = \frac{\sqrt{\sum_{i=1}^{w}(v_1 - v_i')}}{maxDist_{ts}} \qquad (7)$$

where $maxDist_{ts}$ denotes the maximum distance of any pair of objects in the dataset and is used for normalization, as above.

- **Hybrid Distance**: We define a *hybrid* distance measure $dist_h(T, T')$ between two objects T and T', combining both their distances $dist_{sp}(T, T')$ in the spatial domain and $dist_{ts}(T, T')$ in the time series domain. For that purpose, we apply an exponential decay function to decrease the similarity of two time series based on their spatial distance:

$$sim_h(T, T') = sim_{ts}(T, T')\, e^{-\gamma\, dist_{sp}(T, T')} \qquad (8)$$

where $sim_{ts}(T, T') = 1 - dist_{ts}(T, T')$ and γ is the exponesntial decay constant. Then:

$$dist_h(T, T') = 1 - sim_h(T, T') \qquad (9)$$

Accordingly, the hybrid similarity (distance) between two objects T and T' is equal to their standard time series similarity (distance) if they are located at the same position, and it gradually *decreases* (respectively, *increases*) if one is placed farther apart from the other.

3.2.2. Hybrid Query Variants

We are interested in hybrid queries on geolocated time series, i.e., queries that retrieve search results based on *both* spatial distance and time series distance. In these queries, a geolocated time series T_q is given as a reference, and the goal is to identify similar time series to T_q, based on both spatial distance and time series distance.

Different variants of such queries can be derived as follows. First, the query condition can be applied on each distance *independently* or on the hybrid distance defined above. Second, the condition can be a boolean filter (i.e., retrieve all objects with distance lower than θ) or a top-k filter (i.e., retrieve the k objects having the smallest distance). These lead to the query variants outlined below:

- $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$: This query applies two individual boolean filters. It retrieves each time series T having $dist_{sp}(T_q, T) \leq \theta_{sp}$ and $dist_{ts}(T_q, T) \leq \theta_{ts}$. In other words, it retrieves small time series that are located within a radius $\theta_{sp} \cdot maxDist_{sp}$ from T_q's location and their time series dissimilarity to T_q is at most $\theta_{ts} \cdot maxDist_{ts}$.
- $Q(T_q, k, \theta_{ts})$: This query retrieves the k time series closest to T_q's location also having $dist_{ts}(T_q, T) \leq \theta_{ts}$.
- $Q_{bk}(T_q, \theta_{sp}, k)$: This query retrieves the k most similar time series to T_q which are also located within distance $\theta_{sp} \cdot maxDist_{sp}$ from T_q's location.
- $Q_{hb}(T_q, \theta_h, \gamma)$: This query retrieves all time series having hybrid distance to T_q at most θ_h, i.e., $dist_h(T_q, T) \leq \theta_h$.
- $Q_{hk}(T_q, k, \gamma)$: This query retrieves the k time series with the smallest hybrid distance $dist_h(T_q, T)$ to T_q.

Example 3: Figure 14 illustrates an example including two hybrid queries, $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$ and $Q_{kb}(T_q, k, \theta_{ts})$, on a set of geolocated time series $T_1, \ldots, T_9$. In both cases, the reference time series T_q specified by the query is shown in red. Moreover, those time series having dissimilarity to T_q at most θ_{ts} (i.e., T_2, T_4, T_6, T_9) are also shown with red lines. In the first query, only time series within a radius equal to the spatial distance threshold θ_{sp} are retrieved. Thus, the result contains only time series T_2 and T_4, whereas T_6 and T_9 are excluded. The second query for $k = 3$ retrieves the three closest time series to T_q having dissimilarity at most θ_{ts}, i.e., T_2, T_4 and T_6.

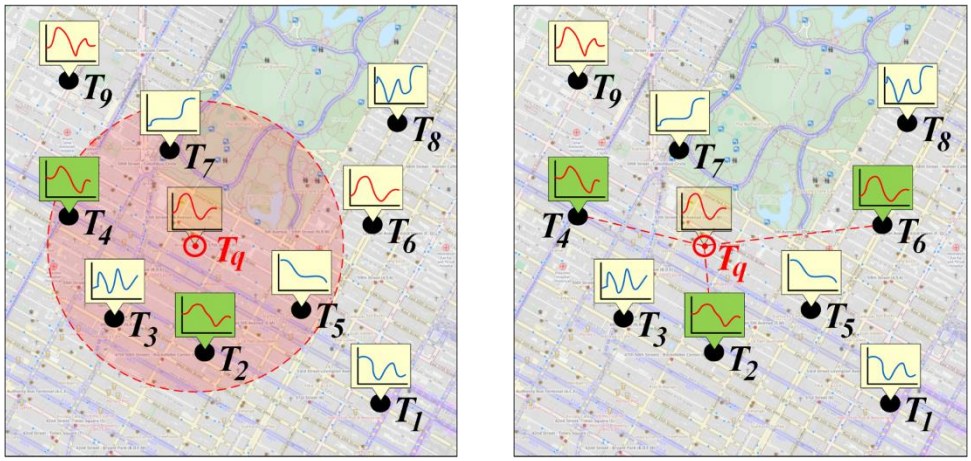


Figure 14: Hybrid queries on geolocated time series.

In the context of DAIAD, the above queries will be used in order to retrieve selected households, as described in the following examples.

Example 4:

- $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$: Retrieve all the households, located within the given radius from "household A" that have similar water consumption time series with it.
- $Q\ (T_q, k, \theta_{ts})$: Retrieve the k nearest households to "household A" that have similar water consumption time series with it.
- $Q_{bk}\ (T_q, \theta_{sp}, k)$: Retrieve the k most similar to "household A" households in terms of water consumption time series that are located within the given radius from it.
- $Q_{hb}\ (T_q\ , \theta_h\ , \gamma)$: Retrieve all the households within the given radius in terms of hybrid distance from "household A".
- $Q_{hk}\ (T_q\ , k, \gamma)$: Retrieve the k nearest households to "household A", in terms of hybrid distance.

3.3. The BTSR-Tree Index

This section outlines the structure of the BTSR-Tree index and presents all the theoretical aspects that lead to its implementation.

3.3.1. Index Structure

In the following, we describe the basic structure of the hybrid index. The initial *naïve* version of the index is first introduced, named *TSR-Tree,* together with the issues that were risen, the solution of which lead us towards the final solution, the BTSR-Tree, which follows.

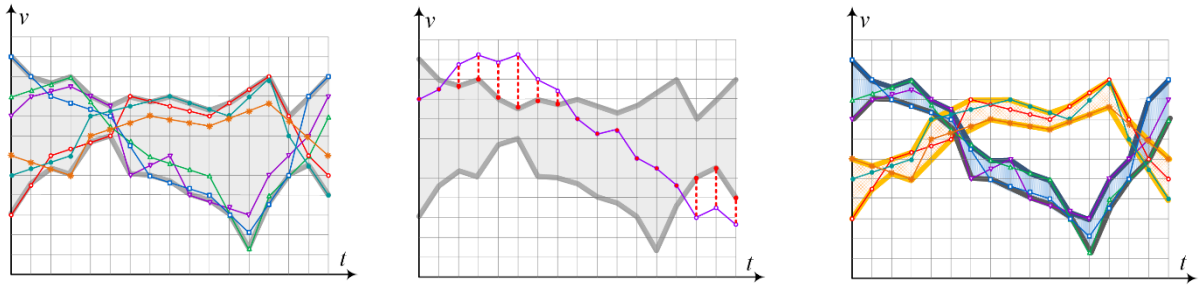
3.3.1.1. TSR-Tree

As mentioned, we initially implemented a naïve version of the index, named *TSR-tree*, which is an *enhanced* R-tree. As in the standard R-tree [Gu84], each node has at least m and at most M entries and stores the MBRs of its children. Additionally, for each child, a node stores a pair of *Minimum Bounding Time Series* (MBTS) that enclose *all* the time series indexed in its subtree. More specifically, this pair consists of an *upper bounding* time series T_{up} and a *lower bounding* time series T_{lo}, respectively constructed by selecting the maximum (for T_{up}) and minimum (for T_{lo}) of values at each time point among all objects indexed therein. Formally, let T_N denote the set of objects contained in the subtree rooted at entry N of a node. Then, the MBTS of N comprises *two* (virtual) time series, $T_{N,up}$ and $T_{N,lo}$ which are derived by selecting the maximum and minimum values, respectively, among all the time series in T_N, i.e.:

$$T_{N,up} = \left\{ \max_{T \in T_N} T[0], \dots, \max_{T \in T_N} T[w-1], \right\} \quad (10)$$

$$T_{N,lo} = \left\{ \min_{T \in T_N} T[0], \dots, \min_{T \in T_N} T[w-1], \right\} \quad (11)$$

Example 5: Figure 15(a) illustrates an example of the MBTS computed from a set of given time series. The latter are represented in the Figure by colored solid lines with different markers. The upper and lower bounding time series are represented by thick grey lines. As can be seen, these enclose the whole (shaded) area where the values of all individual time series lie.



(a) MBTS in TSR-tree (b) Distance bounds with MBTS (c) MBTS in BTSR-tree

Figure 15: Examples illustrating the time series bounds and pruning in TSR-tree and BTSR-tree.

Construction and maintenance of the TSR-tree follow the standard procedures of the R-tree for:

- Data insertion
- Deletion
- Node splitting

Objects (i.e., geolocated time series) are always inserted into leaves. The difference is that, after the R-tree has been built, it is traversed in a reverse *breadth-first* manner and the MBTS of each node are calculated. Moreover, the heuristic for determining where each object will be inserted can be adapted to also account for the similarity of objects in the time series domain, in addition to their spatial distance. Recall that in the standard R-tree, selecting the node where a new object will be inserted is based on finding the entry where

such an insertion incurs the *least possible* enlargement of its MBR. Now, this is extended to also consider the enlargement incurred in that entry's MBTS. Specifically, selecting the appropriate node N should *minimize* the following hybrid *cost function*:

$$cost(T, N) = \lambda \cdot cost_{sp}(T, N) + (q - \lambda) \cdot cost_{ts}(T, N) \qquad (12)$$

where T is the new object for insertion, $cost_{sp}$ and $cost_{ts}$ are functions quantifying the cost of enlarging N's MBR and MBTS, respectively, and λ is a weight parameter determining the relative importance of the two factors. Notice that for $\lambda = 1$ this heuristic behaves *exactly* as in the standard R-tree. When checking a given object T for insertion in node N, we quantify enlargement $cost_{ts}$ of its MBTS as the sum of distances δ_i between T and MBTS at each time point i:

$$\begin{cases} T[i] - T_{N,up}[i], & if\ T[i] > T_{N,up}[i] \\ T_{N,lo}[i] - T[i], & if\ T[i] < T_{N,lo}[i] \\ 0, & if\ T_{N,lo}[i] \leq T[i] \leq T_{N,up}[i] \end{cases} \qquad (13)$$

3.3.1.1.1. TSR-Tree Bounding Tightness

It is apparent that the pruning effectiveness depends on the tightness of these *bounds*. The TSR-tree does not provide any explicit guarantee that time series contained in the same node are highly similar to each other, hence the produced bounds are typically expected not to be sufficiently tight. Further, constructing the aggregate bounds from a set of dissimilar time series, yields bounding time series that are *not* very similar to any of the enclosed ones. These observations are obvious in the example of Figure 15(a) where the constructed MBTS encloses a much larger (grey shaded) area than the one actually occupied by the contained time series, and the resulting bounding time series do *not* closely resemble any of the original ones.

3.3.1.2. BTSR-Tree

To overcome the above issue regarding the tightness of the bounds, we present an optimized version of the TSR-tree, which we call *BTSR-tree*. In a nutshell, the idea is to *bundle* similar time series together in each node, and construct individual MBTS *per bundle*. This allows to derive bounding time series that are tighter and resemble more closely the enclosed ones.

Example 6: Figure 15(c) illustrates this idea using the same example as in Figure 15(a). In this case, the original time series are grouped in two bundles, and the MBTS of each bundle is constructed separately. It is obvious that the resulting bounds are now much tighter, also eliminating much of the dead space within the MBTS and allowing more precise similarity comparisons.

The BTSR-tree index is built similarly to the TSR-tree index. To construct the time series bundles within each node, we rely on k-*means clustering*. To avoid confusion with the top-k predicate in queries, we symbolize with β the number of bundles to be created. The process is performed *bottom-up*, starting from the leaf nodes of the index. In each leaf node, the contained time series are clustered into β bundles. Then, the MBTS of each bundle is computed and stored in the node. Each internal node receives all the MBTS of its children to compute its own β bundles and respective set of MBTS. Thus, the process propagates *upwards*, until reaching the root of the tree. This procedure introduces the following issues:

- **Number of bundles**: The higher the number β of bundles, the *tighter* the resulting bounds. But this also implies that a larger number of MBTS needs to be maintained within each node. Hence, the number of bundles that can be created is limited by *node capacity*.
- **Heterogeneity at higher levels**: As we move from the leaf nodes to the root, a higher number β of bundles is needed, since nodes become increasingly more *heterogeneous* in the time series they contain in their subtree.

To address such issues, we increase the number of bundles *bottom-up* at every tree level by a factor c. Hence, a node at level i has $\beta_i = c \cdot \beta_{i-1}$ bundles; at leaf level, such number β_0 is fixed. At the same time, in order to compensate this increase in terms of node capacity M, we decrease the resolution of the MBTS by the same factor c. The latter is achieved using PAA [KC+01, YF00]. PAA is a common technique that can approximate a time series $T = \{v_1, \ldots, v_w\}$ of length w in to a time series $\overline{T} = \{\overline{v_1}, \ldots, \overline{v_{w'}}\}$ of any arbitrary length $w' \leq w$. In general, each $\overline{v_i}$ is calculated as follows:

$$\overline{v_i} = \frac{w'}{w} \sum_{j=\frac{w}{w'}\cdot(i-1)+1}^{(w/w')i} v_j \qquad (14)$$

In our case, $w' = w/c$. To preserve bounds when applying PAA on an upper (lower) bounding time series T_{up} (T_{lo}), instead of taking the average as in Equation 14, we compute their max (min) values.

3.3.2. Hybrid Node Pruning

For efficient query execution, the goal is to *reduce* the number of node accesses by *pruning* subtrees of the index that cannot contain any results. To do so, we need to establish *lower* bounds for the different types of distances between the query object T_q and any objects contained in the subtree rooted at a node N in the index. In the spatial domain, this bounding $mindist_{sp}(T_q, N)$ is computed as in the case of the standard R-tree, i.e., based on N's MBR. Similarly, in the time series domain, the corresponding bounding distance is derived by N's MBTS. Following Equations 7 and 13, it can be easily seen that this can be calculated as follows:

$$mindist_{ts}(T_q, N) = \frac{\sqrt{\sum_{i=1}^{w} \delta_i^2}}{maxDist_{ts}} \qquad (15)$$

where δ_i represents distances between T_q and MBTS at each time point i, analogously to Equation 13.

Example 7: Figure 15(b) depicts a query time series (the magenta solid line) and the bounds (thick grey lines) in the MBTS of a node. The vertical dashed red lines outside MBTS indicate distances between the query and those bounds contributing to $mindist_{ts}(T_q, N_i)$.

Moreover, given that the objects under a node N are a *subset* of those of its parent N', it follows from the definition of the time series bounds (Equations 10 and 11) that the MBTS of N is *tighter than* (or equal to) the MBTS of N'. From Equation 15, this guarantees that:

$$mindist_{ts}(T_q, N') \leq mindist_{ts}(T_q, N) \qquad (16)$$

So, if $mindist_{ts}(T_q, N')$ is higher than threshold θ_{ts} specified by the query, the *entire* subtree rooted at N' can be pruned altogether. Finally, from Equations 8 and 9 we can observe that hybrid distance $dist_h(T_q, T)$

is monotone with respect both to $dist_{sp}(T_q, T)$ and $dist_{ts}(T_q, T)$. This implies that a minimum hybrid distance bound $mindist_h(T_q, N)$ for the contents of a given node N can be similarly established by combining the individual bounds $mindist_{sp}(T_q, N)$ and $mindist_{ts}(T_q, N)$, i.e.:

$$mindist_h(T_q, N) = 1 - \big(1 - mindist_{ts}(T_q, N)\big) e^{\gamma \cdot mindist_{sp}(T_q, N)} \qquad (17)$$

3.3.3. Hybrid Query Processing

Query processing in the BTSR-tree index follows a similar procedure to the respective processes of evaluating boolean range [Gu84] and kNN queries [HS99] in R-trees. However, this is now extended by utilizing all available bounds $mindist_{sp}$, $mindist_{ts}$ and $mindist_h$, which allows us to prune nodes *simultaneously* in the spatial and time series dimension while traversing the index. Consequently, this reduces the required number of node accesses during evaluation of hybrid queries. In the following, we outline the algorithms for executing each of the query variants presented in Section 3.2.2.

- $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$: To evaluate this query, we recursively traverse the BTSR-tree starting from the root. For each node N, the following two conditions are checked: (a) $mindist_{sp}(T_q, N) \leq \theta_{sp}$ and (b) $mindist_{ts}(T_q, N) \leq \theta_{ts}$. If either of these conditions returns false, N is *pruned*. Once a leaf node is reached, the objects (i.e., geolocated time series) contained therein are *retrieved*, and each one is checked if it qualifies the query criteria. The steps for processing this query are shown in detail in Algorithm 4.

```
1  begin
2      R ← ∅
3      L ← root entries
4      while L ≠ ∅ do
5          N ← L.getNext()
6          if N is not leaf then
7              foreach N' ∈ N.getChildren() do
8                  if mindist_sp(T_q, N') ≤ θ_sp ∧
                   mindist_ts(T_q, N') ≤ θ_ts then
9                      L ← L ∪ {N'.getChildren()}
10         else
11             foreach T ∈ N.getObjects() do
12                 if dist_sp(T_q, T) ≤ θ_sp ∧ dist_ts(T_q, T) ≤ θ_ts then
13                     R ← R ∪ {T}
14             return(R)
```

Algorithm 4: Query $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$

- $Q_{kb}(T_q, k, \theta_{ts})$ and $Q_{bk}(T_q, \theta_{sp}, k)$: These queries combine a boolean filter and a top-k filter. To retrieve top-k results, we employ a *best-first* search approach using a priority queue, as in a typical best-first traversal algorithm [HS99]. Assume that we are evaluating a $Q_{bk}(T_q, \theta_{sp}, k)$ query. Initially, the root entries are retrieved, and their lower spatial distance bound $mindist_{sp}$ is calculated. For a given such entry, if its $mindist_{sp} \geq \theta_{sp}$, then the corresponding subtree is pruned. Otherwise, this entry's $mindist_{ts}$ is calculated, and the entry is pushed to the priority queue according to its $mindist_{ts}$ value in *increasing order*. The process continues recursively, pulling the next item from the queue. If this is a leaf, we retrieve all its objects, calculate their exact spatial

distances $dist_{sp}$, and the ones that are *not* filtered out are pushed to the queue, based on their $mindist_{ts}$. If a pulled item is an object (i.e., geolocated time series), we add it to the result set. The process terminates once k results have been retrieved. Algorithm 5 describes the above procedure in more detail. Evaluating a $Q_{kb}(T_q, k, \theta_{ts})$ is *straightforward*; the treatment of the spatial and the time series dimensions is simply reversed.

```
1  begin
2      R ← ∅
3      Queue ← root
4      while Queue is not empty do
5          X ← Q.pull()
6          if X is a time series then
7              R ← R ∪ {X}
8              if |R| = k then
9                  break
10         else if X is leaf node then
11             foreach T ∈ X.getObjects() do
12                 if dist_sp(T_q, T) ≤ θ_sp then
13                     T.dist ← dist_ts(T_q, T)
14                     Queue.push(T, T.dist)
15         else
16             foreach N ∈ X.getChildren() do
17                 if mindist_sp(T_q, N) ≤ θsp then
18                     N.dist ← mindist_ts(T_q, N)
19                     Queue.push(N, N.dist)
20     return(R)
```

Algorithm 5: Queries $Q_{kb}(T_q, k, \theta_{ts})$ and $Q_{bk}(T_q, \theta_{sp}, k)$

- $Q_{hb}(T_q, \theta_h, \gamma)$: These queries apply a boolean and a top-k filter, respectively. In both cases, this is applied on the hybrid distance $dist_h(T_q, T)$ of the query object T_q to each of the candidate objects T. For $Q_{hb}(T_q, \theta_h, \gamma)$, query evaluation follows the same process as outlined in Algorithm 4. The difference is that instead of checking separately the given distance thresholds on the spatial and the time series distances, a single check is made, comparing the hybrid distances $dist_h$ and $mindist_h$ against the single threshold θ_h. Evaluation of the $Q_{hk}(T_q, k, \gamma)$ query is similar to the one outlined in Algorithm 5. In this case the difference is that there is *no boolean filtering* involved, so objects and nodes are inserted into the priority queue according to their hybrid distance, $dist_h$ and $mindist_h$, respectively. Again, the algorithm terminates once k objects are pulled from the queue; these are the top-k results.

4. Experimental Evaluation

In this section, we present a comparative benchmarking evaluation of the FML-kNN and BTSR-Tree algorithms that were described in the previous sections.

4.1. FML-kNN

This section evaluates the FML-kNN using synthetic data. Further, we apply and evaluate FML-kNN against real-world data in two major cases for the project: (a) forecasting a households' water consumption *simultaneously* for all households within a city, (b) producing predictive analytics from shower events performed by *multiple* households. The experiments were performed on a *pseudo-distributed* environment, setup on a machine located at the University of Peloponnese at Tripolis. It should be noted that, the methodology that is described in the following pages is currently used at DAIAD@Utility for producing the various analytics. More details on the benchmarking environment and the datasets can be found in **Error! Reference source not found.** and Annex: Evaluation datasets.

4.1.1. Experimental setup

4.1.1.1. Metrics

We use four different well known performance measures in order to evaluate the quality of the results obtained by the classifier and the regressor. These performance measures are included in the framework in order to allow expert users to assess the various data analysis tasks. *Accuracy* and *F-Measure* are implemented for classification, while *Root Mean Square Error* (RMSE) and *Coefficient of determination* ($\mathbf{R^2}$) are used to evaluate the quality of regression. A short description of what each of these metrics represents in our experimentation is listed below:

- **Accuracy**. The percentage of *correct classifications* (values from 0 to 100). It indicates the classifier's ability to correctly identify the proper class for each element.
- **F-Measure**. The weighted average of *precision*[9] and *recall*[10] of classifications (values from 0 to 1). Using this metric, we ensure good balance between precision and recall, thus, *avoiding* misinterpretation of the accuracy.
- **Root Mean Squared Error (RMSE)**. Standard deviation between the real and predicted values via regression. This metric has the same unit as the target variable. It provides us with the insight of how close the guessed values are to the real ones.

[9] The fraction of (binary) classifications as 'true' that are correct over the whole dataset.

[10] The fraction of (binary) classifications as 'true' that are correct over the number of elements labelled as 'true'.

- **Coefficient of determination (R^2)**. Indicates the quality of the way the model fits the input data. It takes values from 0 to 1, with 1 being the *perfect fit*. A good fit means that the regressor is able to properly identify the variations of the training data.

4.1.1.2. Parameters

FML-kNN uses a variety of input parameters required by the distributed kNN algorithm, in order to support the classification and regression processes. Regarding the value of the k parameter that was used throughout the experiments (Sections 4.1.2, 4.1.3 and 4.1.4) and due to the fact that the optimal value is problem specific, we performed a separate *cross-validation*[11] evaluation for each use case. The best k parameter is determined in order to support the best balance between *completion time* and *accuracy*.

FML-kNN's core algorithm receives as input a vector of size equal to the dimension of the input dataset, indicating the weight of each feature according to its significance to the problem. Each feature is multiplied with its corresponding weight before the execution of the algorithm in order to perform the required scaling, according to the feature's importance. To automatically determine an optimal feature weighting vector, we execute a *genetic algorithm*[12], which uses a specific metric as cost function (see Section 4.1.1.1). The parameters of the genetic algorithm, such as the size of the *population*, the probability to perform *mutation* or *crossover*, the *elite* percentage of the population and the number of the iterations are currently hard-coded, but in a future extension they will be provided as arguments.

The number of partitions throughout the experimental evaluation is set to 8 and the number of shifts is set to 2. Thus, the level of parallelism of all distributed tasks was set to 16, in order to process the partitions simultaneously.

4.1.2. Benchmarking

We present a comparative benchmarking with similar Apache Spark[13] and Hadoop-based implementations of the probabilistic classifier (the results are similar for the regressor). Hadoop does *not* enable the unification of the sessions because the output cannot be pipelined between the stages and reducers wait the mappers to finish their execution before they start. For this reason and in order to conduct the benchmarking, we also implemented the probabilistic classifier in three sessions for Flink and Spark. As a single session version is not supported in Hadoop, we only implemented the single session version in Spark. FML-kNN as presented has been implemented in a single session version.

They are all executed on a single node HDFS, over a local YARN resource manager (see **Error! Reference source not found.**). This way, each Flink task manager, Spark executor or Hadoop daemon runs on a different YARN

[11] Cross-validation is implemented by iteratively splitting the entire dataset into ten equal parts and then executing the algorithm the same number of times, using a different subset as training set while the rest of the sets, unified, comprise the testing set. Cross-validation outputs the average of a specific metric, across all executions. We used accuracy for classification and RMSE for regression.

[12] A genetic algorithm, optimizes the solution for an optimization problem by iteratively randomly mutating (i.e., applying small changes) a population of solutions, or performing a crossover among them (i.e., combining two or more solutions to a new one). During each iteration, a number of solutions is chosen as the elite population and remains unchanged. At the end of execution, the solution with the best score (i.e., according to a specific metric) is chosen as the final solution.

[13] http://spark.apache.org

container. The comparison was carried out using the *synthetic dataset* (see Synthetic datasets). More particularly, wall-clock time and scalability comparison was carried-out among:

- **FML-kNN (single session).** The main algorithm, presented in the previous section.
- **FML-kNN (three sessions).** A three-session version of FML-kNN, where each stage is executed by a different Flink process.
- **Spark kNN (single session, referred to as S-kNN).** An Apache Spark version with the same architecture as FML-kNN.
- **S-kNN (three sessions).** A three-session version of S-kNN, where each stage is executed by a different Spark process.
- **H-zkNNJ.** An extended version of the algorithm executed in three separate sessions. We extended over this algorithm with the ability to perform probabilistic classification.

A one-session version of the H-zkNNJ algorithm is *not possible*, due to the fact that Hadoop can only execute one map, followed by one reduce operation. A single session requires the three stages to be executed in a *sequential manner*, which is not possible in Hadoop, as it would require mapping to be performed after reducing operations several times.

Despite significant differences in the distributed processing platforms' configuration settings, the different implementations were configured in order to use the same amount of system resources. A maximum of 16 simultaneous tasks (see Section 4.1.1.2) are executed in all cases:

- For Flink and Spark, a total of 4 task managers (one per CPU) and executors respectively were assigned 32768 MB of Java Virtual Machine (JVM) heap memory. Each task manager and executor was given a total of 4 processing slots (Flink) or executor cores (Spark).
- For Hadoop, the parallelism ability was set to 16 by assigning the mappers and the reducers to the same number of YARN containers, with 8192 MB of JVM each. Thus, the total amount of JVM heap memory assigned to each session is always 131072 MB (either 4 x 32768 MB, or 16 x 8192 MB).

4.1.2.1. Wall-clock completion

Table 2 shows the probabilistic classifier's wall-clock completion time of all Flink, Spark and Hadoop versions, run in either three or one sessions, where possible. From the 100M elements of the synthetic dataset, 90% (90M) were used as the training set (S), while the rest 10% (10M) were used as the testing set (R), i.e., the elements we want to classify. It is apparent that:

- Flink-based implementations perform significantly better than the rest, in both three and one-session versions. This is due to Flink's ability to process tasks in a pipelined manner, which allows the *concurrent* execution of successive tasks, thus, gaining in performance by compensating time spent in other operations (i.e., communication between the cluster's nodes).
- For Flink, the total time of the three-session version is similar to the unified one, again due to the pipelined way of execution, which compensates the time lost during HDFS I/O operations during each session.

- The one-session Spark is significantly *faster* than the total wall-clock time of the corresponding three-session setting. This is due to the reduction of the I/O operations on HDFS during the beginning and the end of each session.
- The wall-clock completion time of Spark is slightly *lower* for each stage than Hadoop, confirming Spark's documented better performance over Hadoop.

Version	3 Sessions				1 Session
	1st	2nd	3rd	Total Time	Total Time
F-*k*NN	03m 45sec	24m 59sec	01m 48sec	30m 32sec	30m 25sec
S-*k*NN	07m 12sec	29m 15sec	03m 01sec	39m 28sec	33m 03sec
H-*zk*NNJ	06m 00sec	31m 19sec	05m 54sec	43m 13sec	N/A

Table 2: Wall-clock completion time

4.1.2.2. Scalability

Figure 16 shows the way each version scales in terms of completion time for ten different dataset sizes. The datasets were generated by splitting the synthetic dataset, obtaining datasets containing from 10M (1M testing, 9M training), to 100M (10M testing, 90M training) elements. As illustrated in the figure, the Flink-based implementations exhibit similar performance and scale *better* as the dataset's size increases. However, the unified version, i.e., the one used in FML-kNN, has the advantage of not requiring user input and session initialization between the algorithm's stages. Flink's pipelined execution advantages are apparent if we compare the scalability performance of all the three-session implementations: The I/O HDFS operations cause the Spark and Hadoop versions to scale significantly worse than Flink, which performs similarly to the unified version.

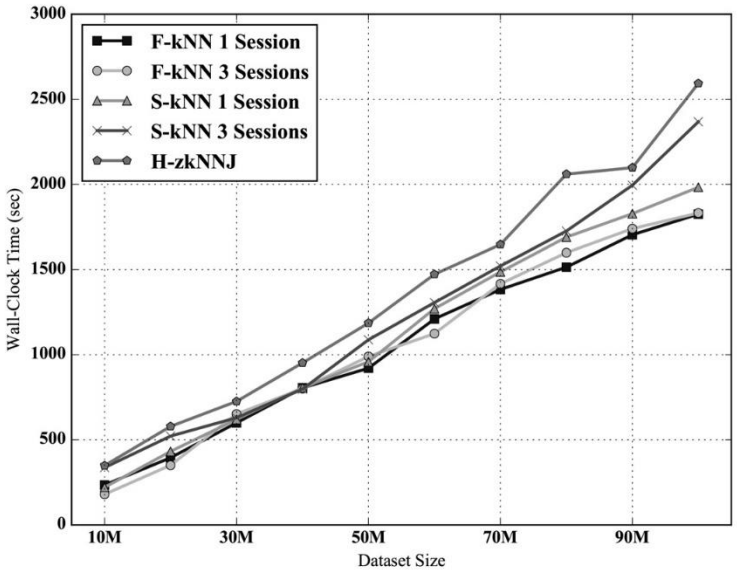


Figure 16: Scalability comparison

4.1.3. Water consumption forecasting

In this section, we use the *Smart Water Meter* (SWM) dataset (see Annex: Evaluation datasets) to apply FML-kNN in *forecasting* future water consumption, based on historical consumption data in the form of time series. We ran FML-kNN over all consumers' data in order to assess the scalability and the prediction precision of our algorithmic framework. We also conducted an experimental evaluation regarding the three SFCs in order to determine which of them better fits our needs regarding prediction accuracy. However, there is a trade-off between the *prediction accuracy* and the *time performance*, we leave the user to select the preferable SFC while executing FML-kNN through the command line.

Using FML-kNN for water consumption forecasting, raises the question whether using a distributed Big Data approach is preferable than standard, centralized, in-process methodologies. Our approach, in order to be able to efficiently forecast consumption in various granularities (i.e., per household) using very large volumes of data, has a trade-off in precision which is a result of the dimensionality reduction applied through the space filling curves. This Big Data-based approach allows us to ensure the *scalability* of the forecasting services for *each individual household* on a city-scale, and the *versatility* of our analytics by supporting on-demand forecasting by experts for *ad hoc consumer groups (e.g., by location, socio-demographics, consumption classes, or arbitrary)*. Compared to standard forecasting methods, our solution is much more complex to devise and implement, but is inherently more *scalable, robust, versatile*, *ubiquitous,* and relevant both for feeding timely interventions inducing changes in consumption behavior at the household level, as well as for the prominent hardware infrastructures of the business sector (*i.e., cloud-based distributed processing frameworks*), thus addressing the actual needs of real-world deployments at the city-scale.

4.1.3.1. Feature extraction

Time series data on water consumption pose several challenges on applying machine learning algorithms for forecasting purposes. Apart from the actual measurement values, one must take into account their *correlation* with previous values, their *seasonality*, the effect of *context factors*, etc. [HC11]. To achieve this, we extracted a total of *nine* temporal and consumption-related features for each data element. We also integrated *five* weather conditions-related features into the dataset. They were obtained via the Weather Underground API[14] by taking into account the date of the timestamp for the area of interest. As mentioned above (see 2.1.2.3), after extracting the features, we sort them in an *ascending* order according to their corresponding water consumption. Then, we *re-label* them with an increasing number according to their new sequence, in order to adapt to the nature of the nearest neighbors algorithm by "placing" similar elements close to each other.

In the following, we present all the features:

- Hour. The hour during which the consumption occurred. In order to adjust to kNN algorithm's nature and due to the fact that the target value is water consumption, we calculated the average consumption of each hour, *sorted* it according to the result and *labelled* it.
- Time Zone. We grouped the hours into four time zones of consumption: 1am - 4am (sleep), 5am - 10am (morning activity), 11am - 7pm (working hours) and 8pm - 12am (evening activity). Similarly, we sorted and labelled according to the average time zone consumption.

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

- **Day of week**. The day of the week from 1 (Monday) to 7 (Sunday). We sorted and labelled according to the average daily consumption.
- **Month**. The month, from 1 (January) to 12 (December). We sorted and labelled according to the average monthly consumption.
- **Weekend**. This is a binary feature which indicates if the consumption occurred during a weekend. We decided to include this feature after noticing differences between average hourly consumption during weekdays and weekends.
- **Customer group.** We run a k-Means[15] clustering algorithm, configured to run for time series data using Dynamic Time Warping (DTW) as a centroid distance metric, on the weekly average *per-hour* consumption for each customer. We determined that ten was the optimal number of clusters, considering the number of customers (1000) and by taking into consideration the clustering quality. We measured the latter using the Davies-Bouldin index metric (see Section 4.1.1.1). Similarly, we sorted and labelled according to per cluster average consumption.
- **Customer ranking**. For each hour, we calculated the average hourly consumption of each customer and sorted according to it. Then, we labelled each element according to its ranking in the sorted list.
- **Customer class**. This feature represents customer groups of one and up to four classes according to their monthly average consumption, i.e. "*environmentally friendly*", "*normal*", "*spendthrift*", "*significantly spendthrift*".
- **Season**. The season, from 1 (winter) to 4 (autumn). Similarly, we sorted and labelled according to consumption per season.
- **Temperature**. The temperature during the hour (Celsius scale).
- **Humidity**. The relative humidity during the hour (percentage).
- **Weather conditions**. The weather conditions during the hour, integer labelled (i.e. cloudy = 1, sunny = 2, overcast = 3, etc.).
- **Continuous rainfall**. Amount of hours of continuous rainfall.
- **Continuous heat**. Amount of hours of continuous heat (above 25°C).

Each element also contained *two target variables*, being the *exact* meter reading at each timestamp and a binary flag, indicating whether consumption occurred during that hour or not (i.e., if the meter reading was positive or zero). In the future, we will include further features that affect water demand, such as important and local events (e.g., national holidays, festivals, football games, etc.).

4.1.3.2. Procedure

The consumption forecasting is performed for *all* the households (1000) in the dataset for the last two available weeks. The procedure is consisted of the following two processes:

[15] *k*-Means clustering partitions *n* elements into *k* clusters, in which each element belongs to the cluster with the nearest *centroid*, which is calculated by averaging the elements of each cluster.

- **Classification.** We first execute F-kNN probabilistic classifier, with the testing set (R) comprising of the last *two weeks* of water consumption for each household, in order to obtain the possibility of whether consumption will occur or not, during each hour. The rest of the dataset is used as the training set (S). We perform binary classification, obtaining an intermediate dataset indicating whether or not consumption will occur for each testing element.
- **Regression.** Using the hours during which we predicted that water will be consumed, we run F-kNN regressor, obtaining a full predicted time series result of water consumption for each user.

Before we execute each algorithm, we determine the optimal scale vector using the *genetic approach* (see Section 4.1.1.2). Also, in order to choose the optimal k parameter for both algorithms, we employed a ten-fold cross-validation approach. The k value that achieved the best balance between completion time and result quality was 15, for both F-kNN probabilistic classifier and regressor.

FML-kNN supports three SFC-based solutions for reducing the dimensionality of the input data to just one dimension, namely the z-order, Hilbert and Grey code curves. We evaluated the completion time and approximation quality of each SFC, in order to select the best choice which balances time performance and approximation accuracy. Table 3 presents the metric and time performance related results of F-*k*NN probabilistic classifier and regressor for each SFC, which we obtained from running the algorithms using the *ten-fold* cross-validation.

Curve	Classification			Regression		
	Accuracy	F-Measure	Wall-clock time	RMSE	R^2	Wall-clock time
z-order	70.24%	0.775	1m 20sec	18.86	0.64	0m 59sec
Hilbert	70.54%	0.78	1m 32sec	18.69	0.66	1m 15sec
Grey-code	70.4%	0.777	1m 25ec	18.81	0.64	1m 5sec

Table 3: Space filling curves' performance

All three SFCs experience *similar* performance. Hilbert curve scores higher in all metrics as expected, however only slightly. Consequently, we choose the z-order curve to perform water consumption forecasting tasks, as it exhibits better time performance due to its decreased calculation complexity.

4.1.3.3. Results

Among all the households and for each hour during the last two available weeks, the classifier correctly predicted the 74.54% (F-Measure: 0.81) of the testing set's target variables, i.e., the hours during which some consumption occurred for the two-week period. For these specific hours, the regressor achieved a RMSE score of 19.5 and a Coefficient of determination score of 0.69. The results were combined into a single file, forming the complete time series of the dataset's last two weeks' water consumption prediction for all the users.

Figure 17 shows four users' consumption prediction versus the actual one, during four different days. The prediction for user #4 was close to the actual consumption. The results seem to follow the real values, but are not able to properly follow the observed ones. This indicates that it is *rather difficult* to accurately predict a single user's future water consumption, due to possible *random* or *unforeseen* events during a day, a fact which justifies the rather large RMSE score. For example, consumptions higher that 20 liters during an hour

(e.g., user #3 around 6:00) could indicate a shower event, while larger consumptions (>50 liters) over more than one hour could suggest usage of the washing machine or dish washer (e.g., user #3 from 16:00 to 20:00), along with other activities.

In order to assess the generalization of the results, we calculated the average RMSE of the hour and volume of the peak consumption during each day, as well as the average RMSE of the total water use per day, for all the predictions. The rather high errors, (8.89 hours, 28.9 liters and 132.23 liters respectively), confirm that it is a hard task to accurately predict random daily events. However, despite the difficulty, our algorithms' predictions are able to mostly follow the overall behavior during most days (e.g., user #3 and user #1).

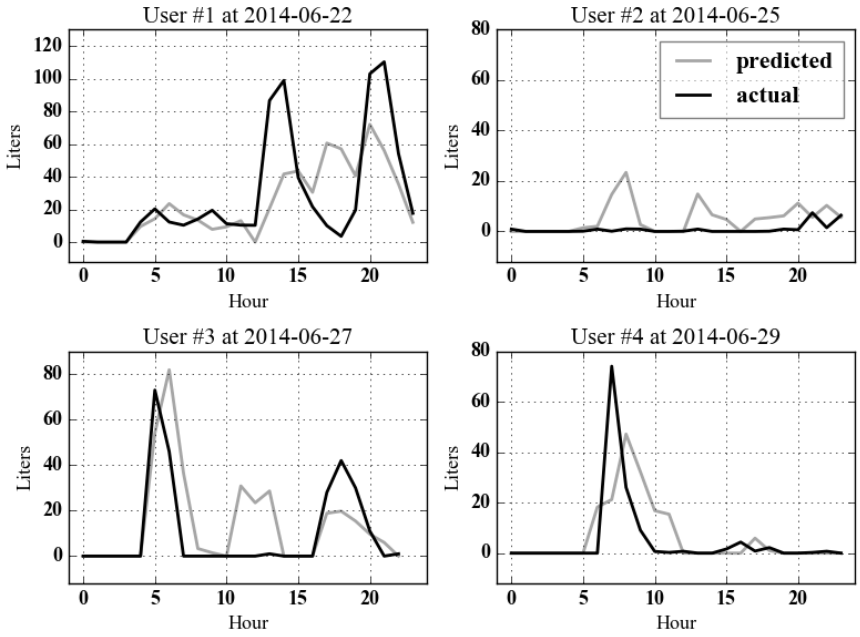


Figure 17: Personalized hourly water consumption prediction

The results are particularly interesting if we *aggregate* the total predicted consumption of all users during each hour. The two upper diagrams in Figure 18 illustrate this case for two different days. The two top sub-figures illustrate FML-kNN's performance for two different days (22/06/2014 and 25/06/2014) after we have aggregated the individual predictions into a single, overall, aggregate prediction against the real aggregate consumption. The two bottom sub-figures depict the prediction for the same days after performing a *centralized* (on a single machine) kNN based forecasting on already aggregated (pre-aggregated) consumption data. It is apparent that our algorithms' predictions are able to properly follow the real aggregated consumption during each hour. This indicates that the effect of random events is *raised* when we predict the total hourly water consumption of a larger number of users. The two lower diagrams present the same prediction performed by feeding our algorithms with already aggregated hourly consumption data. While the pre-aggregated dataset's predictions appear to less diverge from the actual values in certain points, the non-aggregated results better assemble the actual overall behavior. This is due to the fact that the non-aggregated dataset contains user-specific features and the task to perform predictions based on user similarity yields more accurate predictions.

A reasonable question, at this point, could be why we prefer to focus on producing individual consumption prediction, which is computationally intense and requires Big Data technology than a centralized method that operates on aggregates of water consumption, or on a reduced set of *representative users*, (in the context of *data thinning/set cover* methods). As mentioned above, providing widely available (i.e., accessible by everyone) water forecasting information has a possible effect on altering water consumption behaviour. We, thus, favour this forecasting granularity level in order to feed our interventions to our participants. Also, one would argue that, individual water consumption is rather *random*, and such a fact would significantly enhance the difficulties of our efforts. Our analysis results (please refer to deliverable D7.3: Trials Evaluation and Social Experiment Results) suggest *otherwise*: per household water consumption *is not random* and is correlated with several characteristics, such as the apartment size, the income, the household location, number of household members, etc. It is thus clear that, in a scenario where such information would be available for *every* household, such an individual consumption forecasting would yield even better results.

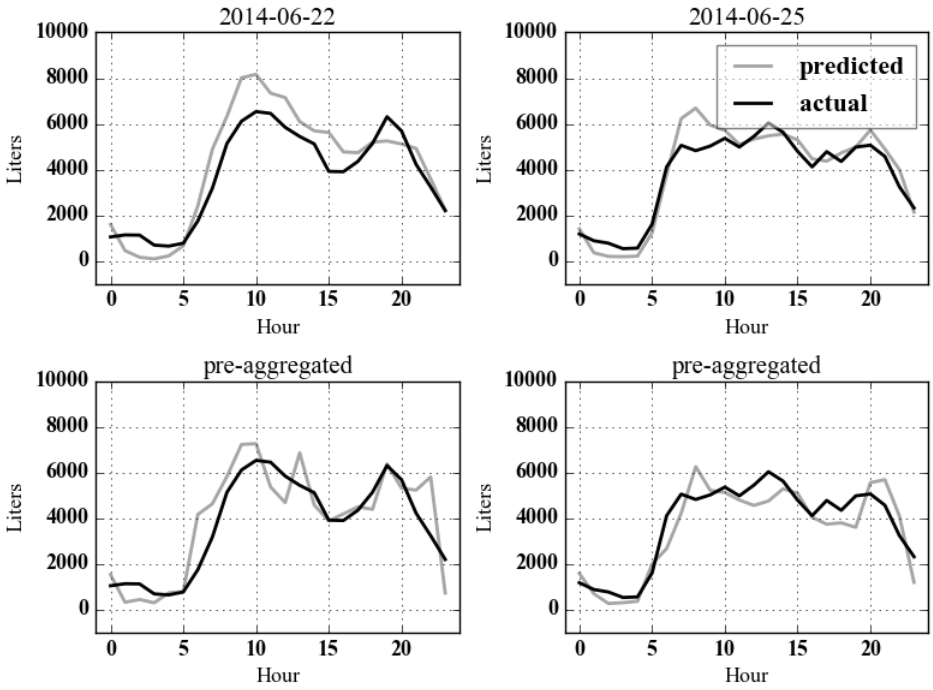


Figure 18: Aggregated hourly water consumption prediction. The predicted values seem to correctly follow the general trend of consumption

Finally, in order to illustrate the use of the class probabilities that F-*k*NN probabilistic classifier outputs, we split the hourly consumption into five static classes (minimum, low, normal, high, maximum [16]), depending on the amount of water that was spent during each hour. We then performed a multiclass classification to obtain the probabilities that each household's consumption will belong to each of the classes. Figure 19 shows a visualization of the classifier's output for a specific household. The bars indicate the household's water consumption probability during each hour.

16 minimum: 0 - 1 liters, low: 1 - 4 liters, normal: 4 - 12 liters, high: 12 - 26 liters, maximum: > 26 liters

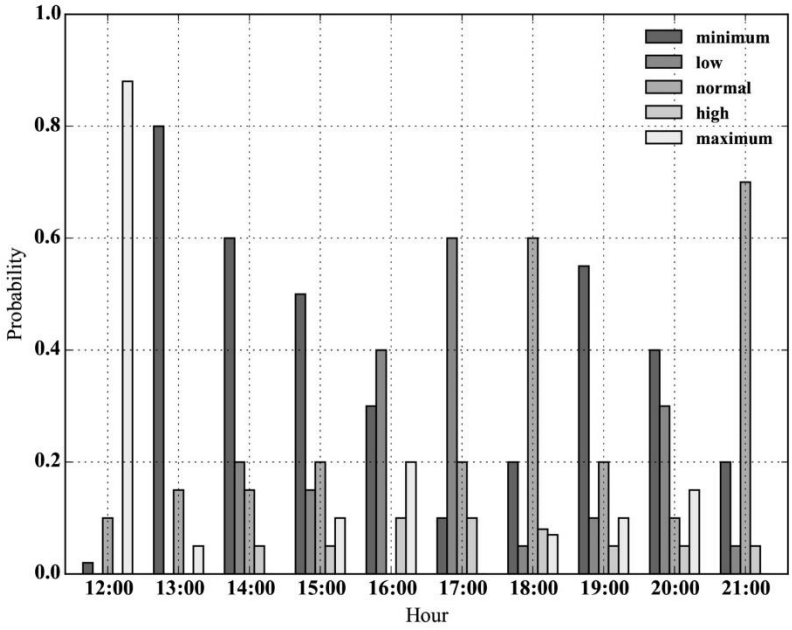


Figure 19: Probability of each consumption class per hour

4.1.4. Shower predictive analytics

In this section, we apply FML-*k*NN on the *shower consumption* dataset (see Annex: Evaluation datasets) to produce various predictive analytics. This dataset also includes anonymized demographics information related with the consumers. The analytics include the prediction of the *sex*, *age* and *income* of the person that generates a shower event. As the elements of the dataset in these cases can be classified in two groups based on the classification rule (i.e. sex: male or female, age: less than 35 or not, income: less than 3000 CHF or not), thus we used binary classification.

4.1.4.1.Feature extraction

Initially, we extracted the proper features and target variables from the dataset. Regarding the latter, for the case of sex, we used the shower events for which we knew whether the person was male or female (binary classification), i.e., households with only one, or of the same sex members. Consequently, each element consisted of all the smart meter measurements (shower stops, break time, shower time, average temperature, and average flow rate) as features and the sex as the target variable. Due to the dataset's rather small size, the age and income prediction was also performed by *binary classification*, i.e., determine whether the person that generates a shower event is of age less than 35 years or not, or has income less than 3000 CHF or not. The features of each element were the same in all three cases.

4.1.4.2. Procedure

We performed binary classification of the shower events for each of the target variables, using the probabilistic classifier. Ten-fold cross-validation was also used in this case. The latter, similarly to the previous case, helped us determine the optimal value of k parameter, which was set to 10.

4.1.4.3. Results

The results obtained by the classification are illustrated in Figure 20. The classifier achieved cross-validation accuracy of 78.6%, 64.7% and 61.5% for sex, age and income prediction respectively.

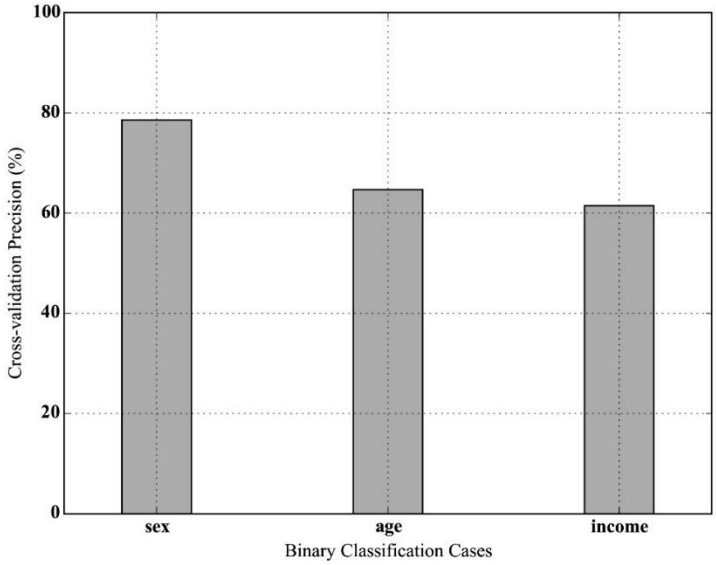


Figure 20: Cross-validation accuracy for sex, age and income prediction

4.2. BTSR-Tree Hybrid Index

In this section, we present a detailed experimental evaluation of the BTSR-Tree hybrid index. The evaluation includes the comparison of the BTSR-Tree and the naïve TSR-Tree with a basic implementation of an R-Tree index, in various real-world scenarios and for different parameters. First, we describe our experimental setup, the datasets and the applied parameterizations, and then we discuss the results.

4.2.1. Experimental Setup

4.2.1.1. Datasets

We have applied our methodology against *four* real-world datasets. These are selected from various application domains and have different characteristics, in order to assess the performance of our approach when dealing with diverse types of geolocated time series. The datasets, which are further described in Annex: Evaluation datasets, include:

- **Alicante SWM Dataset (Water)** – see SWM dataset
- **NYC taxi dropoffs (Taxi)** – see NYC taxi drop-offs dataset
- **Flickr geotagged photos (Flickr)** – see Flickr geotagged photos dataset
- **UK historical crime data (Crime)** – see UK historical crime dataset

Regarding the SWM dataset, we calculated the *average weekly* water consumption for each household. Since the resulting dataset had a rather small number of entries (approx. 1000) and in order to test the index at its limits, we used the resulting weekly time series as seeds, in order to *synthetically increase* the size of the dataset to 200,000, by introducing some random variations in their location and pattern. The characteristics of each dataset are listed in Table 4.

Dataset	Area ($\boldsymbol{km^2}$)	Number of locations	Length of time series	Default Query Thresholds		
				θ_{sp}	θ_{ts}	θ_h
Water (DAIAD)	114	200000	168	0.15	0.175	0.15
Taxi	2500	417960	168	0.2	0.001	0.0025
Flickr	Earth	414967	96	0.25	0.0005	0.001
Crime	392000	362215	76	0.3	0.01	0.02

Table 4: Datasets used in the experiments

4.2.1.2. Index Parameters

For all indices, we set the minimum ($\boldsymbol{m}$) and maximum ($\boldsymbol{M}$) number of entries stored in each node to 60 and 200, respectively. To insert time series in the TSR-tree and the BTSR-tree in the same manner as in the R-tree, the weight parameter $\boldsymbol{\lambda}$, used in Equation 12 to select the node with the least cost during an insertion, is set to 1. This implies that only the *spatial cost* is considered during insertion. This way, the performance benefits observed in the experiments are only due to the pruning conditions. Finally, for the BTSR-tree, we fix the number of bundles to $\beta_0 = 5$ for its leaf nodes; factor $\boldsymbol{c}$, specifying the increase (decrease) rate of the number of bundles (respectively, time series resolution) at each higher level in the tree hierarchy, is set to $c = 2$.

4.2.1.3. Query Parameters

The query parameters involve the different types of distance thresholds for queries involving boolean filtering, namely spatial ($\boldsymbol{\theta_{sp}}$), time series ($\boldsymbol{\theta_{ts}}$) and hybrid ($\boldsymbol{\theta_h}$), as well as the number $\boldsymbol{k}$ of results to return for queries involving top-$\boldsymbol{k}$ filtering. Values to these parameters are set differently for each dataset, based on their characteristics; default values are shown in Table 4. Moreover, for queries involving hybrid distance, we *fix* the exponential decay constant $\boldsymbol{\gamma}$ to 0.025 for the Water and Taxi, 0.001 for the Flickr and 0.05 for the Crime dataset, to better reflect the *spatial distribution* and *coverage* of each particular dataset.

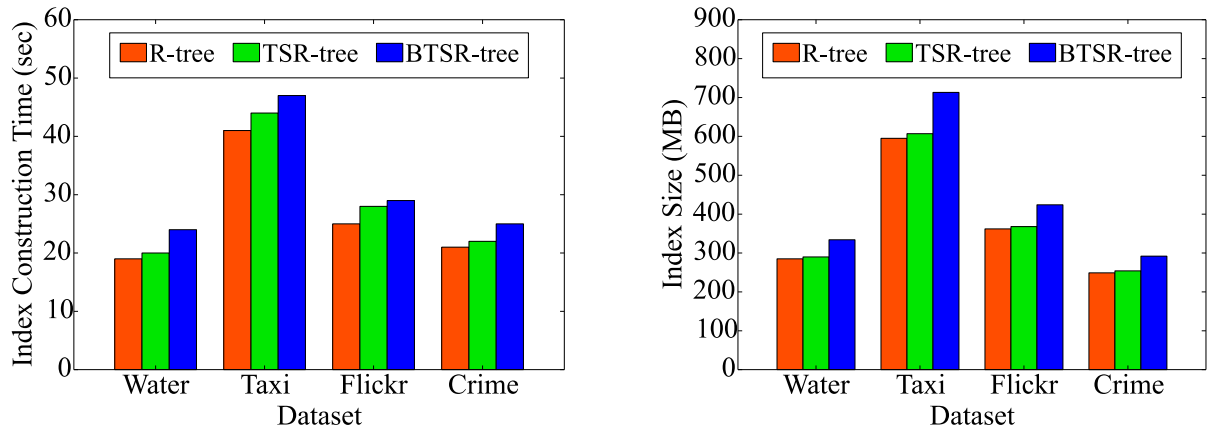
4.2.1.4. Evaluation Setting

To evaluate the effectiveness of our proposed hybrid indices, we compare their performance to the standard R-tree, used as *baseline*. In the R-tree, query evaluation is done by traversing the index according to the spatial predicate of the query, retrieving a set of intermediate results that satisfy the spatial condition, and then *filtering out* candidates according to the time series predicate to produce the final result set. Performance

comparison is done by measuring the portion of the tree nodes accessed by each method, since this is the dominant performance factor for each index. Moreover, for each index, we measure the time required for its construction as well as its size. The implementations of all indices are currently *in-memory*. Query workloads were created by randomly picking 500 geolocated time series separately for each dataset. Comprehensive results from these experiments are presented next.

4.2.2. Index Construction Time and Size

Figure 21 shows the index *construction time* and the resulting index *size* (i.e., memory footprint) for each dataset. Construction time of the TSR-tree index is slightly higher than that of the R-tree, and the same holds for the index size. This is natural, because after building the spatial part of the index, it has to be *traversed* in order to calculate the MBTS of each node, which incurs additional cost both for computing and for storing it in each node. Compared to the TSR-tree, building the BTSR-tree index is even *more costly*, since it incurs an additional time and space overhead to compute the time series bundles in each node and maintain the MBTS of each bundle. Still, these costs are not considered significant. Even for the largest dataset (Taxi), which contains 417, 960 time series and has an original size of 153 MB, the index construction time is around 43 seconds for TSR-tree and 45 seconds for BTSR-tree, while the index size is around 610 MB and 710 MB, respectively. Note that the extra space for the BTSR-tree is needed for storing the bounds for each of the progressively multiple bundles in internal nodes.



(a) Index construction time

(b) Index size

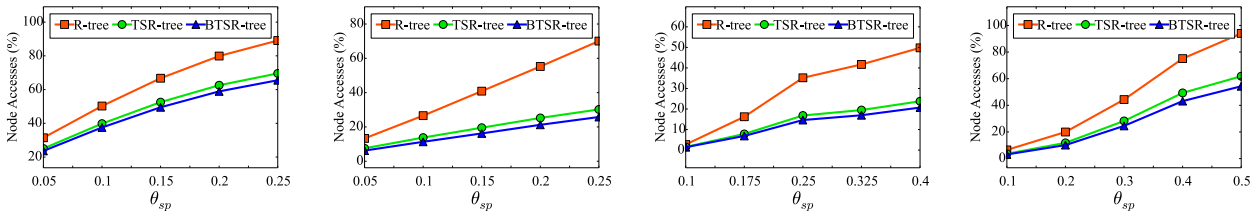
Figure 21: Comparison of index construction time and size for each dataset

4.2.3. Query Performance

In the following, we present an evaluation of our methodology, comparing the percentage of nodes accessed by each index in each query. We compare performance of the standard R-tree method with the TSR-tree and the BTSR-tree, using *all* four datasets, for *all* supported queries. All measurements are average cost per query in each particular workload.

4.2.3.1. Query Q_{bb}

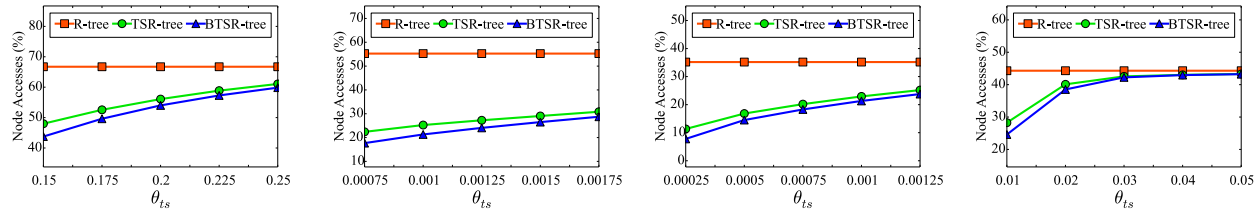
Figure 22 illustrates the performance of the Q_{bb} query on each dataset for varying spatial distance threshold (θ_{sp}). Both the TSR-tree and the BTSR-tree clearly outperform the standard R-tree in all datasets, with *gains* in performance increasing even further as the threshold θ_{sp} increases. The standard R-tree needs to access a significant amount of nodes as it can only prune in the spatial domain. As a result, it performs increasingly worse than the TSR-tree and the BTSR-tree. Due to its tighter bounds, the BTSR-tree manages to prune more nodes than the TSR-tree, especially for larger spatial threshold values.



(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 22: Query $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$ with varying spatial distance threshold θ_{sp}.

Obviously, increasing the time series threshold (Figure 23) has absolutely no impact on the performance of the standard R-tree. On the contrary, the node accesses required by the TSR-tree and the BTSR-tree are much *lower*. Nevertheless, these also increase with more relaxed θ_{ts}, asymptotically reaching those of the R-tree. Performance of the BTSR-tree is better for lower values of θ_{ts}, as it allows *more* pruning thanks to the tighter bounds of bundles. It is apparent that the sensitivity of the threshold *heavily depends* on the dataset. Even slightly increasing θ_{ts} in the Taxi and Flickr datasets (Figure 23(b) and Figure 23(c)), significantly affects performance of both the TSR-tree and the BTSR-tree, as they are more *sparse* than the Water and Crime datasets (Figure 23(a) and Figure 23(d)), with a large number of time series having many zero values. Consequently, even a small increase in this threshold causes a larger amount of nodes to be probed.



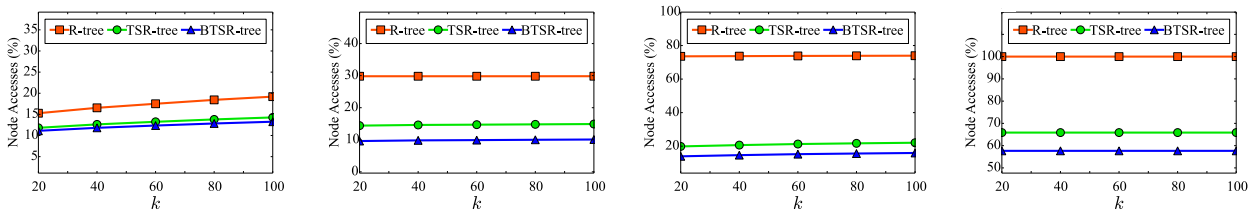
(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 23: Query $Q_{bb}(T_q, \theta_{sp}, \theta_{ts})$ with varying spatial distance threshold θ_{ts}.

4.2.3.2. Query Q_{kb}

Performance of Q_{kb} for varying number k of results is depicted in Figure 24. In all cases, both the TSR-tree and the BTSR-tree cope *better* compared to the standard R-tree. Varying the number k of results does *not* significantly affect performance, as nearby elements tend to have similar time series values and all required results are found in close distance. This is an interesting insight, especially for the Water dataset, which

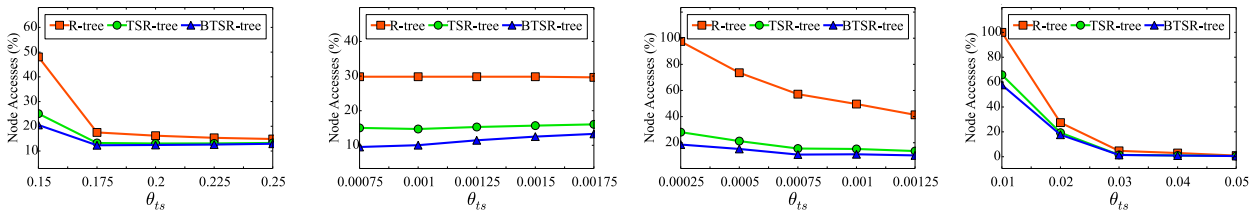
suggests that neighboring households tend to have *similar* water consumption. The R-tree performs really poor in the Crime dataset (Figure 24(d)), requiring a full traversal (100% of its nodes), as the sought number of results *cannot* be found. This is not the case with the TSR-tree and the BTSR-tree, respectively accessing 65% and 58% of their nodes due to their effective pruning in the time series domain.



(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 24: Query $Q_{kb}(T_q, k, \theta_{ts})$ with varying number k of results.

Increasing the time series threshold θ_{ts} (Figure 25) in Q_{kb}, *improves* performance of the R-tree and it is sometimes competitive to that of the TSR-tree and the BTSR-tree, as *more* nearby elements qualify as results and are thus quickly found. However, this is strongly influenced by dataset characteristics. For the Taxi dataset (Figure 25(b)), node accesses in the R-tree are *similar* for different values of θ_{ts}, but performance for the BTSR-tree slightly *decreases* with higher threshold values. Indeed, relaxing this threshold effectively allows more tolerance in the similarity of time series and thus, extra nodes need to be visited for checking.

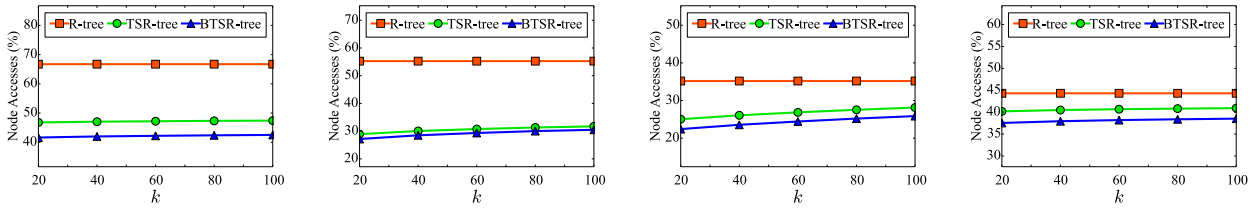


(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 25: Query $Q_{kb}(T_q, k, \theta_{ts})$ with varying spatial distance threshold θ_{ts}.

4.2.3.3. Query Q_{bk}

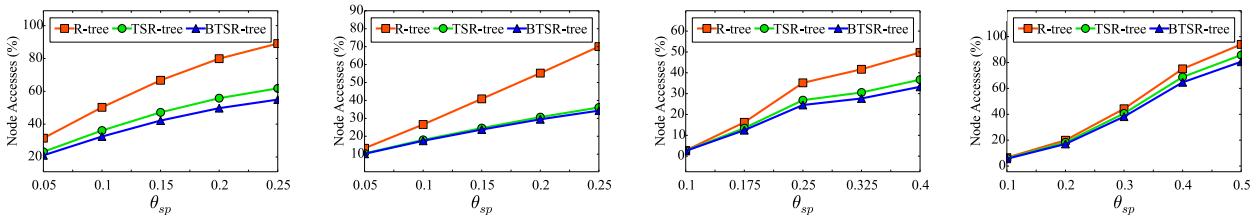
Varying the number k of results in the Q_{bk} query (Figure 26), shows similar behavior to the Q_{kb} query. Similar time series are located close to each other, so k results are *quickly* obtained once the first qualifying time series is retrieved. The R-tree is always significantly *outperformed* by the TSR-tree and BTSR-tree; the effect is less apparent in the Crime dataset (Figure 26(d)).



(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 26: Query $Q_{bk}(T_q, \theta_{sp}, k)$ with varying number k of results.

Figure 27 illustrates the performance of the Q_{bk} query for varying spatial distance threshold (θ_{sp}). The significantly better performance of the TSR-tree and the BTSR-tree is also apparent here, with the difference getting more pronounced with *larger* thresholds (i.e., wider radii). This advantage is less manifest in the Crime dataset (Figure 27(d)) because the applied thresholds cover increasingly *larger* spatial areas (practically the entire dataset for θ_{sp} = 0.5), thus more nodes need to be examined in order to identify the k results. In the Taxi dataset, the performance improvement of the BTSR-tree compared to the TSR-tree is *smaller* due to data sparsity, which *diminishes* the pruning effect of bundles in the nodes of the BTSR-tree.

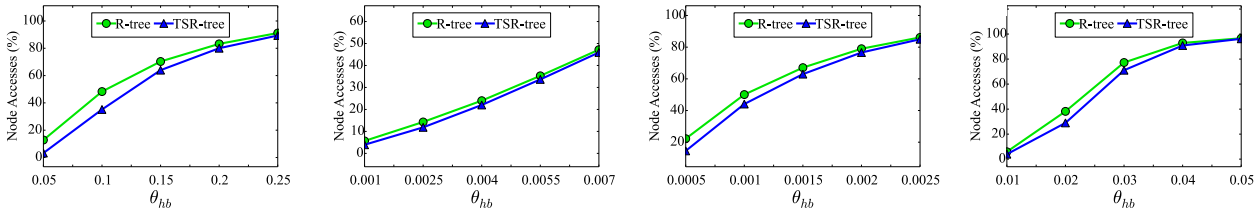


(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 27: Query $Q_{bk}(T_q, \theta_{sp}, k)$ with varying spatial distance threshold θ_{sp}.

4.2.3.4. Queries Q_{hb} and Q_{hk}

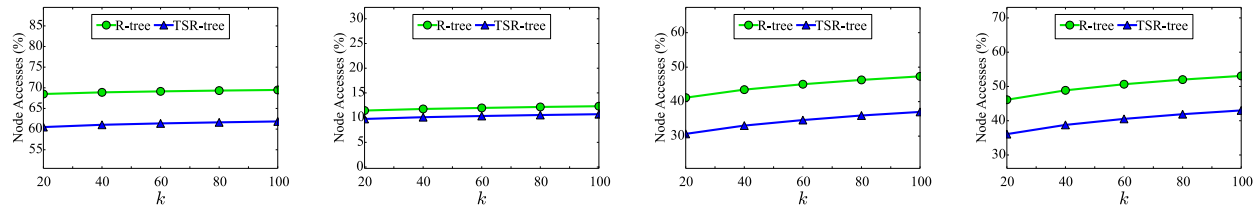
First, note that such hybrid queries *cannot* be possibly applied on the R-tree at all. Regarding the Q_{hb} query, Figure 28 plots performance for varying hybrid distance threshold (θ_{hb}). The BTSR-tree fares *better* than the TSR-tree in all cases. The effect is more intense with smaller θ_{hb} values, since the tighter bundles in the BTSR-tree allow more effective pruning. For each dataset, divergence in performance between the two indices *largely de*pends on variance among closely located time series. For instance, taxi dropoffs exhibit similar patterns locally, so the derived bounds also tend to be similar, *diminishing* the pruning power of both indices as shown in Figure 28(b). Instead, when nearby time series have many diverse patterns, the BTSR-tree becomes *more* effective by allocating them to distinct bundles and offering considerable performance gain as for the Water dataset (Figure 28(a)).



(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 28: Query $Q_{hb}(T_q, \theta_h, \gamma)$ with varying spatial distance threshold θ_h.

Finally, Figure 29 depicts the performance of Q_{hk} for varying number k of results. BTSR-tree always outperforms TSR-tree with a margin of *more* than 10% node accesses on average, again with the exception of the Taxi dataset (Figure 29(b)), as mentioned before.



(a) Water (b) Taxi (c) Flickr (d) Crime

Figure 29: Query $Q_{hk}(T_q, k, \gamma)$ with varying number k of results.

5. Annex: Evaluation datasets

5.1. Synthetic datasets

5.1.1. SWM dataset

Using a Big Data generator, (part of the BigDataBench benchmark suite[17]) we generated a synthetic dataset based on the SWM dataset's format after the proper feature extraction. The synthetic dataset consists of 100 million records of the following format:

- An alphanumeric identifier.
- 9 integer features with 10 decimals, from 1 to 99.
- 2 features as target variables.
    - One floating point with 10 decimals from 1 to 99.
    - One binary.

An example of several records is the following:

```
Alanah,13,1,6,10,1,1,5,405,5,75.3193100497,1
Peg,22,3,3,10,3,1,5,380,5,51.2148987257,1
Patch,5,3,6,7,2,1,6,480,7,32.8837968085,1
Quality,7,2,1,7,2,1,9,467,9,31.3787360197,1
Cayla,13,1,4,4,1,1,8,476,7,96.5476215768,1
Oralie,5,1,6,8,1,1,5,822,2,19.4084972708,1
```

5.1.2. Amphiro dataset

Using the same Big Data generator, we created a second synthetic dataset based on the shower related features of the Amphiro a1 dataset. The synthetic dataset consists of 15 million records of the following format:

- An incremental identifier representing the household.
- An incremental identifier representing the shower.
- 5 floating point features of various ranges.
- 1 binary feature as a target variable.

An example of a several records is the following:

[17] http://prof.ict.ac.cn

2234,33,12.21,62.54,3.57,75.23,0.4,1

2234,34,11.23,32.12,1.66,85.92,0.1,0

2234,35,15.19,34.65,1.32,83.22,0.15,1

2236,1,1.23,3.23,2.12,26.71,0.87,0

2236,2,6.18,8.86,5.87,58.21,0.94,0

5.2. Real-World datasets

5.2.1. SWM dataset

The Smart Water Meter dataset contains hourly time series of 1000 Spanish households' water consumption, measured with smart water meters. It covers a time interval of a whole year, i.e., from the 1st of July 2013 to the 30th of June 2014 (a total of 8.600.000 records). The records have the following format:

- An alphanumeric household identifier.
- A timestamp including the date and time.
- A SWM measurement in litres.

Due to possible SWM related hardware or other issues, several measurements were missing. We handled this issue by equally distributing the difference in consumption between the current and last correct measurement to the in-between missing values. To avoid negatively affecting the results, we removed possible outliers, being users that did not consume any water for more than 40 days during a year which may indicate permanent absence, or they continuously consume water (i.e. indication of possible leakage). An example of several records is the following:

C12FA151955;08/06/2016 03:07:17;112370;

C12FA151955;08/06/2016 02:07:17;112369;

C12FA151955;08/06/2016 01:07:17;112369;

C12FA151955;07/06/2016 23:46:26;112368;

C12FA151955;07/06/2016 22:46:26;112366;

5.2.2. Amphiro dataset

The amphiro dataset includes shower events along with demographic information from 77 Swiss households collected with the Amphiro a1 device. The records have the following format:

- An integer household identifier.
- Total number of showers per household.
- An incremental shower identifier.

- Duration of the shower in seconds.
- Duration of interruptions in between one shower (e.g. while soaping) in seconds.
- Water consumption in liters per minute.
- Number of male residents living in the household.
- Number of female residents living in the household.
- Number of residents living in the household and aged between 0 and 5 years.
- Number of residents living in the household and aged between 6 and 15 years.
- Number of residents living in the household and aged between 16 and 25 years.
- Number of residents living in the household and aged between 26 and 35 years.
- Number of residents living in the household and aged between 36 and 45 years.
- Number of residents living in the household and aged between 46 and 55 years.
- Number of residents living in the household and aged between 56 and 65 years.
- Number of residents living in the household and aged between 66 and 75 years.
- Number of residents living in the household and aged over 75 years.
- Whether the costs for water consumption are included in the rent or not.
- Whether the costs for heating energy are included in the rent or not.
- Number of residents using the monitored shower.
- Number or residents having long hair.
- Gender of the survey taker.
- Age of the survey taker.
- Nationality of the survey taker.
- Education of the survey taker.
- Number of adults living in the household.
- Number of children living in the household.
- Form of housing.
- Monthly net income of the household in CHF (Fr).

An example of several records is the following (consecutive commas indicate no value):

> *2640,2,47,47,0,1591,0,304,37.5,11.4645,2,1,1,,,,,,,2,,0,0,2,0,male,65+,Switzerland,apprenticeship,2,no child, rental apartment,81-115 m2,7000 - 7999 Fr.*
>
> *2640,2,47,34,0,216,0,33,37.5,9.16667,2,1,1,,,,,,,2,,0,0,2,0,male,65+,Switzerland,apprenticeship,2,no child,rental apartment,81-115 m2,7000 - 7999 Fr.*

2640,2,47,3,0,31,0,5,37.5,9.67742,2,1,1,,,,,,,2,,0,0,2,0,male,65+,Switzerland,apprenticeship,2,no child,rental apartment,81-115 m2,7000 - 7999 Fr.

2640,2,47,4,0,1406,0,244,37.5,10.4125,2,1,1,,,,,,,2,,0,0,2,0,male,65+,Switzerland,apprenticeship,2,no child,rental apartment,81-115 m2,7000 - 7999 Fr.

5.2.3. NYC taxi drop-offs dataset

This dataset contains time series extracted from yellow taxi rides in New York City during 2015. The original data corresponding timestamps for each ride. For each month, we generated time series by applying a uniform spatial grid over the entire city (cell side was 200 meters) and counting all drop-offs therein for each day of the week at the time granularity of one hour. Thus, we obtained the number of drop-offs for 24 × 7 time intervals in every cell, which essentially captures the weekly fluctuation of taxi destinations there. Without loss of generality, the centroid of each cell is used as the geolocation of the corresponding time series. The records have the following format:

- An alphanumeric cell identifier.
- The Cartesian coordinates of the geolocated time series.
- The time series of the drop-offs during each day.
- An example of several records is the following:

468624,302100,60900,5,8,3,5,0,0,2,1,0,0,1,0,2,1,2,0,1,1,5,2,0,2,3,3,2,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,5,1,1,3,2,2,2,0,0,0,0,1,0,0,0,0,1,0,0,1,1,1,0,3,2,0,3,4,4,2,5,3,1,1,0,1,0,1,0,1,0,1,1,0,1,0,0,1,3,1,4,1,0,3,9,2,0,3,1,1,0,2,0,0,4,2,2,0,1,0,2,0,7,3,3,0,4,1,3,2,1,2,0,0,0,0,0,0,0,0,2,1,0,2,2,1,3,2,5,2,1,1,3,4,3,3,2,0,0,0,0,1,1,1,0,2,4,1,5,3,4,5,1,5,2,8

468625,302100,61100,10,7,9,4,4,0,0,0,1,1,0,2,6,4,8,2,5,8,7,7,5,5,3,9,2,3,0,2,1,2,0,0,3,0,2,0,1,2,1,1,1,2,2,5,5,3,4,4,5,0,3,0,0,0,2,1,0,0,0,1,0,4,3,4,3,3,3,4,4,5,4,3,5,3,2,1,1,0,0,0,2,1,3,0,2,2,1,3,5,6,8,6,7,0,2,8,8,2,0,0,0,0,0,2,1,0,1,1,4,0,1,2,3,3,1,4,7,6,3,8,7,5,3,2,0,0,1,1,0,3,1,2,0,4,1,1,4,5,8,7,4,9,10,8,11,7,6,6,4,1,1,2,1,1,3,1,3,7,3,5,6,6,4,6,5,13,6,3

The second and third numbers are the coordinates and the rest of the numbers comprise the time series.

5.2.4. Flickr geotagged photos dataset

This dataset contains time series data extracted from geolocated Flickr images between 2006 and 2013 over the entire planet. In order to get meaningful geolocated time series, we partitioned the space by a uniform grid of 7200 × 3600 cells (each one spanning 0.05 decimal degrees in each dimension) and we counted the number of photos contained in every cell each month, excluding cells with no data at all (e.g., in the oceans). Each time series conveys the visits pattern (in terms of number of photos taken per month) of that region over this period. The records have the following format:

- An alphanumeric cell identifier.

- The Cartesian coordinates of the geolocated time series.
- The time series of the photos taken during each month.
- An example of several records is the following:

> *20717219,6.975,50.975,2,4,2,6,5,32,15,119,136,47,4,30,53,26,30,70,20,88,55,216,0,5,8,29,63,38,12,10,65,10,127,42,21,1,93,8,10,2,393,30,46,70,27,42,4,74,1,3,3,22,36,4,20,14,57,109,0,10,11,25,16,9,7,36,36,60,35,169,25,134,190,70,28,28,27,64,54,62,60,226,35,19,14,12,66,5,4,34,26,172,43,14,62,3,2,3,79,8,93,29*
>
> *20717220,6.975,51.025,0,0,0,3,0,0,0,0,0,5,0,2,7,6,2,10,0,0,12,22,0,1,3,0,0,0,2,4,0,5,6,6,0,1,0,2,0,0,0,0,1,1,0,0,1,0,0,0,3,11,1,2,2,7,1,0,0,0,0,2,2,0,0,0,28,81,0,1,0,6,9,2,0,0,1,5,0,0,60,0,0,0,0,0,0,10,2,0,19,0,2,2,3,1,0,1,3,0,0,7,9*

The second and third numbers are the coordinates and the rest of the numbers comprise the time series.

5.2.5. UK historical crime dataset

This dataset contains time series representing the temporal variation in the number of crime incidents reported across England and Wales over 76 months (December 2010– March 2017). From the original data, we generated time series over a grid with cell size 200 meters. For each month, we counted incidents having their location within each cell. The records have the following format:

- An alphanumeric cell identifier.
- The Cartesian coordinates of the geolocated time series.
- The time series of the crimes occurred during each month.
- An example of several records is the following:

> *2890,505500,183100,3,10,3,1,4,12,5,6,6,2,8,4,5,9,2,6,2,3,6,8,1,6,1,1,1,1,3,0,1,2,1,1,2,4,3,0,1,2,0,6,3,3,1,2,5,2,8,4,1,0,5,2,1,2,1,4,2,0,0,3,0,1,0,3,4,1,1,4,1,3,6,3,0,2,0,2*
>
> *2891,505500,183300,3,3,0,0,2,1,3,3,2,2,2,5,4,5,4,5,2,1,3,6,8,11,0,3,5,3,0,2,2,3,5,4,9,1,4,3,0,5,2,1,3,4,6,2,4,3,6,0,0,2,1,2,1,5,3,2,4,6,10,0,4,6,2,4,0,17,3,5,1,6,4,7,2,2,1,5*

The second and third numbers are the coordinates and the rest of the numbers comprise the time series.

6. Annex: Implementation details

6.1. FML-kNN

The FML-kNN framework was developed in Java v1.7, using the Flink API[18] and the InteliJ Idea IDE[19]. For comparison reasons, we also developed a Spark version of the framework's probabilistic classifier and extended the Hadoop-based H-z*k*NNJ algorithm to also perform classification.

FML-kNN is consisted of the following classes (packages are in italic font):

- *mainAPP:* This package contains the necessary methods to start the execution.
    - `Setup`: This class contains the main method, sets up the global configuration from the given parameters and executes the program, either in genetic scale optimization, or in normal mode.
    - `FML_kNN`: This class initiates the Flink environment and executes the transformations after reading the proper data from HDFS. Whether regression or classification will be executed, according to the global configuration is determined here. The same stands for cross-validation or normal mode.
- *tools:* This package contains several classes containing methods that are necessary for the execution of the program and the experimentation.
    - `Functions`: Several generic purpose methods needed in various parts of the program.
    - `ExecConf`: The global configuration is described by this class. It is passed as an argument anywhere it is needed.
    - `CurveRecord`: Class that holds the SFC value for each record, its identifier, the file of origin (R or S), the shift number and the target variable (class or value). It is communicated between the transformations where necessary.
    - `KnnRecord`: Class that describes a nearest neighbor, holding the record identifier, its coordinates (i.e., features) and its distance to the lookup element.
    - `Zorder`: This class contains methods for the forward and inverse calculation of the z-values.
    - `Gorder`: This class contains methods for the forward and inverse calculation of the g-values.

[18] http://flink.apache.org

[19] https://www.jetbrains.com/idea/

- `Horder`: This class contains methods for the forward and inverse calculation of the h-values.

- *`transformations:`* This package contains all the extended Flink transformations used in the program.
  - `MapPhase1`: This class scales the features accordingly (for both R and S datasets), shifts them if necessary, calculates the z-values, g-values or h-values of each record and passes everything to the next transformation (MapPhase1Sampling).
  - `MapPhase1Sampling`: This class samples the datasets and passes everything to the next transformation (ReducePhase1).
  - `MapPhase2`: This class forwards the records of the transformed datasets (both R and S, shifted or not) to the proper reducer described by the next transformation (ReducePhase2)
  - `ReducePhase1`: This class calculates the partitioning ranges from the sampled datasets and continues to the next transformation (MapPhase2)
  - `ReducePhase2`: This class retrieves the kNNs for each element in each partition and passes everything to either of the next two transformations.
  - `ReducePhase3Classification`: This class calculates the final kNNs for each element, performs the final probabilistic classification and stores the results on HDFS.
  - `ReducePhase3Regression`: This class calculates the final kNNs for each element, performs the final regression and stores the results on HDFS.

The execution is invoked by executed the jar executable with the following command:

```
<flink_path>/bin/flink run -m yarn-cluster -yn
<number_of_task_managers> -yjm <java_heap_for_job_manager> -ytm
<java_heap_for_task_managers> -ys
<number_of_slots_per_task_manager> -p <level_of_parallelism>
FML_kNN-1.0-flink-fat-jar.jar <k> <number_of_shifts> <dimension>
<number_of_classes> <epsilon> <number_of_partitions>
<z_g_or_h_SFC> <classification_or_regression> <cross_validation>
```

Parameters:

- `<flink_path>`: Path to the Flink directory.
- `<number_of_task_managers>`: Number of Flink task managers to be created for the execution.
- `<java_heap_for_job_manager>`: Java heap space assigned to the Flink job manager.
- `<java_heap_for_task_managers>`: Java heap space assigned to each task manager.
- `<number_of_slots_per_task_manager>`: Number of parallel slots in each task manager. It is usually set equal to the number of CPUs in the task manager.

- <level_of_parallelism>: The level of parallelism (number of concurrent jobs) for the execution.
- <k>: The number of nearest neighbors for each element.
- <number_of_shifts>: The number of random shifts of the datasets.
- <dimension>: The number of dimensions (i.e., features) of the datasets.
- <number_of_classes>: The number of classes for the classification.
- <epsilon>: This number (from 0 to 1, usually << 1) determines the frequency of sampling the datasets.
- <number_of_partitions>: The number of partitions to be created.
- <z_g_or_h_SFC>: Determines whether z-values, g-values, or h-values should be used (number 1, 2, or 3 respectively).
- <classification_or_regression>: Determines whether probabilistic classification (1) or regression (2) should be executed.
- <cross_validation>: Binary argument, true if cross-validation should be performed.

The screenshots below show an example of the framework's execution. Initially, we connect to the remote server using SSH (Figure 30).

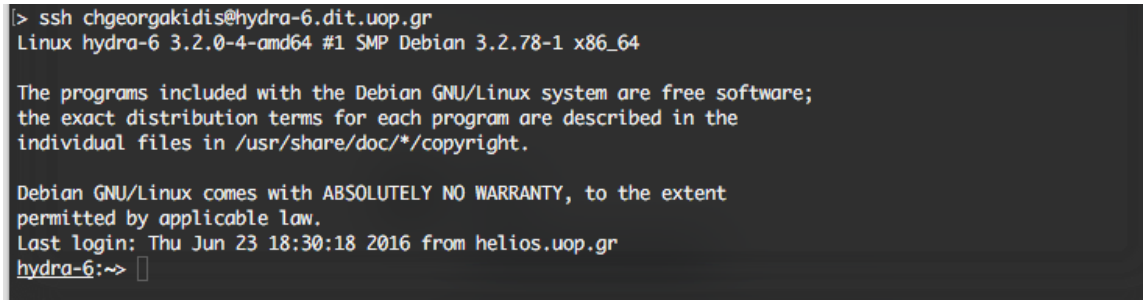


Figure 30: Screenshot showing the connection to the server

```
hydra-6:~/Files/F-kNN/FML_kNN> ../../Frameworks/flink-0.10.2/bin/flink run -m yarn-cluster -yn 4 -yjm 8192 -ytm 32768 -ys 16 -p 16 target/FM
L_kNN-1.0-flink-fat-jar.jar 15 2 9 2 0.003 8 2 1 false
YARN cluster mode detected. Switching Log4j output to console
14:47:19,268 INFO  org.apache.hadoop.yarn.client.RMProxy                     - Connecting to ResourceManager at /0.0.0.0:8032
14:47:19,485 INFO  org.apache.flink.client.FlinkYarnSessionCli               - No path for the flink jar passed. Using the location of c
lass org.apache.flink.yarn.FlinkYarnClient to locate the jar
The YARN cluster has 64 slots available, but the user requested a parallelism of 16 on YARN. Each of the 4 TaskManagers will get 4 slots.
14:47:19,498 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Using values:
14:47:19,500 INFO  org.apache.flink.yarn.FlinkYarnClient                     -      TaskManager count = 4
14:47:19,500 INFO  org.apache.flink.yarn.FlinkYarnClient                     -      JobManager memory = 8192
14:47:19,501 INFO  org.apache.flink.yarn.FlinkYarnClient                     -      TaskManager memory = 32768
14:47:20,632 INFO  org.apache.flink.yarn.Utils                               - Copying from file:/home/local/chgeorgakidis/Frameworks/fl
ink-0.10.2/lib/flink-dist-0.10.2.jar to hdfs://localhost:9000/user/chgeorgakidis/.flink/application_1466162028550_0007/flink-dist-0.10.2.jar
14:47:21,621 INFO  org.apache.flink.yarn.Utils                               - Copying from /home/local/chgeorgakidis/Frameworks/flink-0
.10.2/conf/flink-conf.yaml to hdfs://localhost:9000/user/chgeorgakidis/.flink/application_1466162028550_0007/flink-conf.yaml
14:47:21,649 INFO  org.apache.flink.yarn.Utils                               - Copying from file:/home/local/chgeorgakidis/Frameworks/fl
ink-0.10.2/conf/logback.xml to hdfs://localhost:9000/user/chgeorgakidis/.flink/application_1466162028550_0007/logback.xml
14:47:21,674 INFO  org.apache.flink.yarn.Utils                               - Copying from file:/home/local/chgeorgakidis/Frameworks/fl
ink-0.10.2/conf/log4j.properties to hdfs://localhost:9000/user/chgeorgakidis/.flink/application_1466162028550_0007/log4j.properties
14:47:21,708 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Submitting application master application_1466162028550_0
007
14:47:21,747 INFO  org.apache.hadoop.yarn.client.api.impl.YarnClientImpl     - Submitted application application_1466162028550_0007
14:47:21,747 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Waiting for the cluster to be allocated
14:47:21,750 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:22,754 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:23,758 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:24,762 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:25,765 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:26,769 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:27,774 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:28,777 INFO  org.apache.flink.yarn.FlinkYarnClient                     - Deploying cluster, current state ACCEPTED
14:47:29,781 INFO  org.apache.flink.yarn.FlinkYarnClient                     - YARN application has been deployed successfully.
14:47:29,789 INFO  org.apache.flink.yarn.FlinkYarnCluster                    - Start actor system.
14:47:31,081 INFO  org.apache.flink.yarn.FlinkYarnCluster                    - Start application client.
YARN cluster started
JobManager web interface address http://hydra-6:8088/proxy/application_1466162028550_0007/
Waiting until all TaskManagers have connected
```

Figure 31: Deployment of the cluster

After the command is issued the Flink environment is initiated and the cluster is deployed according to the given arguments (Figure 31).

```
All TaskManagers are connected
06/24/2016 14:47:57     Job execution switched to status RUNNING.
06/24/2016 14:47:57     CHAIN DataSource (at execute(FML_kNN.java:166) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap
(FlatMap at execute(FML_kNN.java:175))(1/16) switched to SCHEDULED
06/24/2016 14:47:57     CHAIN DataSource (at execute(FML_kNN.java:166) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap
(FlatMap at execute(FML_kNN.java:175))(1/16) switched to DEPLOYING
06/24/2016 14:47:57     CHAIN DataSource (at execute(FML_kNN.java:166) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap
(FlatMap at execute(FML_kNN.java:175))(2/16) switched to SCHEDULED
06/24/2016 14:47:57     CHAIN DataSource (at execute(FML_kNN.java:166) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap
(FlatMap at execute(FML_kNN.java:175))(2/16) switched to DEPLOYING
06/24/2016 14:47:57     CHAIN DataSource (at execute(FML_kNN.java:166) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap
(FlatMap at execute(FML_kNN.java:175))(3/16) switched to SCHEDULED
06/24/2016 14:47:57     CHAIN DataSource (at execute(FML_kNN.java:166) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap
(FlatMap at execute(FML_kNN.java:175))(3/16) switched to DEPLOYING
06/24/2016 14:47:57     CHAIN DataSource (at execute(FML_kNN.java:166) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap
(FlatMap at execute(FML_kNN.java:175))(4/16) switched to SCHEDULED
```

Figure 32: Execution of the program

After all the Flink task managers are connected, the program is executed. Figure 32 shows Flink debugging output during the execution where the first mapping transformations are scheduled after reading the proper files from HDFS.

Upon finishing, the results for classification are stored on HDFS in the following format:

> *XYZ|Result C|A:0.06|B:0.03|C:0.71|D:0.08|E:0.12*

Here, the element with identifier XYZ is classified to class C, which has the highest probability. Similarly, for regression:

> *XYZ|Result: 19.244469*

Here, the value estimation of element with identifier XYZ is 19.244469.

6.2. FML-SP

FML-SP was developed in Java v1.7, using the Flink API and the IntelliJ Idea IDE. It is consisted of the following classes (packages are in italic font):

- *clustering*: This package contains the classes that implement the k-means clustering.
    - kMeansClustering: This class executes the for k-means clustering after receiving the time series dataset to be clustered.
- *clustering.transformations*: This package contains all the extended Flink transformations used for the k-means clustering.
    - CentroidAccumulator: This transformation sums the time series per hour and counts them according to an appended count variable.
    - CountAppender: This transformation appends a count variable to each element.
    - CentroidAverager: This transformation computes the new centroid from the above time series sum and count.
    - InitialCentroidCalculator: This transformation generates the initial centroids for the execution.
    - SelectNearestCenter: This transformation determines the closest cluster center for an element (a time series)
- *savingspotential*: This package contains classes that implement the FML-SP algorithm.
    - SavingsPotential: This class initiates the Flink environment and executes the transformations after reading the proper data from HDFS.
    - Setup: This class implements the main method for the application
- *savingspotential.transformations*: This package contains all the extended Flink transformations used for the savings potential calculation.
    - *AvgDailyTS*: This transformation calculates the average per day of week consumption in the form of time series.
    - *AvgWeeklyTS*: This transformation calculates the average weekly consumption time series for each given household.
    - *CalculateSP*: This transformation calculates the final savings potential per cluster of households (calculated with k-means)
    - *CalculateWaterIQ*: This transformation calculates the final WaterIQ value for each household.
- *tools:* This package contains several classes containing methods that are necessary for the execution of the program and the experimentation.

- `Functions`: Several generic purpose methods needed in various parts of the program.
- `ExecConf`: The global configuration is described by this class. It is passed as an argument anywhere it is needed.
- `Element`: This class implements an element for the FML-SP algorithm, comprised by an identifier and the time series.

The execution is invoked similarly to FML-kNN as described in FML-kNN. Upon finishing, the results for classification are stored on HDFS in two files, containing the final savings potential and WaterIQ score of each household. The savings potential file has the following format:

> *Cluster5;11;60234;76.1645%*

Here, the group of households named Cluster5 has 60234 liters as savings potential for November (month 11), which is the 76.1645% of the total savings potential that all groups of households can achieve during that month.

Finally, the WaterIQ file has the following format:

> *10;Cluster1;XYZ;A;-1384.00*

Here, for October (month 10), the user with SWM id XYZ that belongs to the group of households named Cluster1, has spend less water equal to 1384 liters, compared to the rest of the households within the same group and has a WaterIQ score A.

6.3. FML-RF

FML-RF is being developed in Java v1.7, using the Flink API and the IntelIJ Idea IDE. It is consisted of the following classes (packages are in italic font):

- *`fmlrf`*: This package contains the necessary methods to start the execution.
  - *`FML_RF`*: initiates the Flink environment and executes the transformations after reading the proper data from HDFS.
  - *`MainApp`*: This class contains the main method of the app and initializes all necessary parameters.
- *`transformations:`* This package contains all the extended Flink transformations used in the application.
  - `MapTransform`: This transformation scales the features accordingly (for both R and S datasets), shifts them if necessary, calculates the z-values, g-values or h-values of each record and passes everything to the next transformation (MapSampling).
  - `MapSampling`: This transformation samples the datasets and passes everything to the next transformation (ReduceComputeRanges).

- `ReduceComputeRanges`: This transformation calculates the partitioning ranges from the sampled datasets and continues to the next transformation (MapPartition).
- `MapPartition`: This transformation forwards the records of the transformed datasets (both R and S, shifted or not) to the proper reducer described by the next transformation (ReduceDecisionTrees).
- `ReduceDecisionTrees`: This transformation calculates the proper decision trees that comprise the local Random Forest of each partition and forwards them to the next transformation (ReduceMergeTrees).
- `ReduceMergeTrees`: This transformation merges the received decision trees and merges them accordingly to form the final Random Forest.

- *`tools`:* This package contains several classes containing methods that are necessary for the execution of the program and the experimentation.
  - `Functions`: Several generic purpose methods needed in various parts of the program.
  - `ExecConf`: The global configuration is described by this class. It is passed as an argument anywhere it is needed.
  - `Element`: This class implements an element for the FML-RF algorithm, comprised by an identifier, a value, a partition number and a label.
  - `Zorder`: This class contains methods for the forward and inverse calculation of the z-values.
  - `Gorder`: This class contains methods for the forward and inverse calculation of the g-values.
  - `Horder`: This class contains methods for the forward and inverse calculation of the h-values.

FML-RF is currently under construction and details regarding its execution cannot be documented for now.

6.4. BTSR-Tree Hybrid Index

The BTSR-Tree hybrid index was developed in Java 1.7 using the IntelliJ Idea IDE. It is consisted of the following classes (packages are in italic font):

- *`kmeans`*: This package contains classes that implement the k-means clustering algorithm.
  - `Cluster`: This class implements a k-means cluster, comprised by an identifier, its points and a centroid.
  - `kMeans`: This transformation executed the k-means clustering algorithm.
- *`rtsindex`:* This package contains all the classes that comprise the BTSR-Tree hybrid index.

- `Entry`: This is an entry of the index, i.e., a geolocated time series.
- `MainApp`: This class executes the application and initializes all the necessary parameters.
- `Node`: This class implements a node of the BTSR-Tree with all the necessary functionality.
- `Query`: This class implements the queries that are supported by the hybrid index.
- `RTSIndex`: This class implements the BTSR-Tree hybrid index containing all the necessary functionality for insertion, deletion and updating.
- `HybridTSData`: This class implements a geolocated time series. It is comprised of a n identifier, the raw time series and the spatial coordinates.

- *`tools:`* This package contains several classes containing methods that are necessary for the execution of the program and the experimentation.
  - `ElementDistance`: This class is used in several priority queues in order to order the elements by a predefined distance (either, time series, spatial or hybrid distance)
  - `ExecConf`: The global configuration is described by this class. It is passed as an argument anywhere it is needed.
  - `ExecQueries`: This class contains the methods that execute the queries.
  - `Experiment`: This class contains all the methods that were used during the experimentation phase.
  - `Functions`: Several generic purpose methods needed in various parts of the program.

The BTSR-Tree index is currently under construction and details regarding its execution cannot be documented for now.

7. References

[Ab08] Abbas OA. Comparisons Between Data Clustering Algorithms. International Arab Journal of Information Technology (IAJIT). 2008 Jul 1;5(3).

[BF14] Balassi M., Fora G., The Flink Big Data Analytics Platform [pdf], 2014. Retrieved from http://www.slideshare.net/GyulaFra/flink-apachecon?from_action=save

[BK04] C. Böhm, F. Krebs, The *k*-nearest neighbor join: Turbo Charging the KDD process, Knowledge and Information Systems 6 (6) (2004) 728–749.

[CC+13] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. PVLDB, 6(3):217–228, 2013.

[CD+11] F. Chen, J. Dai, B. Wang, S. Sahu, M. Naphade, C. Lu, Activity Analysis Based on Low Sample Rate Smart Meters, Proceedings of ACM SIGKDD, 2011, pp. 240–248.

[CF99] K. Chan and A. W. Fu. Efficient time series matching by wavelets. In ICDE, pages 126–133, 1999.

[CH+11] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. In CIKM, pages 423–432, 2011

[CJ+09] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. PVLDB, 2(1):337–348, 2009.

[CK+15] Chatzigeorgakidis G., Karagiorgou S., Athanasiou S., Skiadopoulos S., A MapReduce Based *k*NN Joins Probabilistic Classifier, Proceedings of the 2015 IEEE International Conference on Big Data, pp. 952-957, 2015.

[CP+10] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. iSAX2.0: Indexing and mining one billion time series. In ICDM, pages 58–67, 2010.

[CS+06] Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In SIGMOD, pages 277–288, 2006.

[CS+14] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. Knowl. Inf. Syst., 39(1):123–151, 2014.

[CS+17] Chatzigeorgakidis G., Skoutas D., Patroumpas K., Athanasiou S., Skiadopoulos S., Indexing Geolocated Time Series Data, SIGSPATIAL '17 International Conference on Advances in Geographic Information Systems (under review).

[CW+10] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In SSDBM, pages 87–95, 2010.

[DB79] Davies, David L., and Donald W. Bouldin. "A cluster separation measure." *IEEE transactions on pattern analysis and machine intelligence* 2 (1979): 224-227.

[DC94] Berndt, Donald J., and James Clifford. "Using dynamic time warping to find patterns in time series." KDD workshop. Vol. 10. No. 16. 1994.

[FH+08] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In ICDE, pages 656 – 665, 2008.

[Gr95] A. Graps. An introduction to wavelets. IEEE Comput. Sci. Eng., 2(2):50–61, 1995.

[Gu84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In SIGMOD, pages 47–57, 1984.

[GX+11] J. Gou, T. Xiong, Y. Kuang, A Novel Weighted Voting for k-Nearest Neighbor Rule, JCP6(5) 833–840, 2011.

[Ha10] A. Haar. Zur theorie der orthogonalen funktionensysteme. Mathematische Annalen, 69(3):331 – 371, 1910.

[HC11] L. A. House-Peters, H. Chang, Urban water demand modeling: Review of concepts, methods, and organizing principles, Water Resources Research 47 (5), 2011.

[HH+07] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In SSDBM, page 16, 2007.

[HM79] Hartigan, John A., and Manchek A. Wong. "Algorithm AS 136: A k-means clustering algorithm." *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979): 100-108.

[HS99] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. ACM Trans. Database Syst., 24(2):265–318, 1999.

[IM98] P. Indyk, R. Motwani, Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality, Proceedings of ACM STOC, 1998, pp. 604–613.

[KC+01] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. Knowl. Inf. Syst., 3(3):263–286, 2001.

[KK11] S. Kashyap and P. Karras. Scalable kNN search on vertically stored time series. In SIGKDD, pages 1334–1342, 2011.

[KM+13] E. Kermany, H. Mazzawi, D. Baras, Y. Naveh, H. Michaelis, Analysis of Advanced Meter Infrastructure Data of Water Consumption in Apartment Buildings, Proceedings of ACM SIGKDD, 2013, pp. 1159–1167.

[Ko90] T. Kohonen, The Self-Organizing Map, in *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464-1480, Sep 1990.

[KP13] Kodinariya, Trupti M., and Prashant R. Makwana. "Review on determining number of Cluster in K-Means Clustering." *International Journal* 1.6 (2013): 90-95.

[La00] J. Lawder, Calculation of Mappings Between One and n-Dimensional Values Using the Hilbert Space-Filling Curve, Tech. rep., Birkbeck College, University of London, 2000.

[LC+12] Li B., Chen X., Li M.J., Huang J.Z., Feng S. "Scalable Random Forests for Massive Data". In: Tan PN., Chawla S., Ho C.K., Bailey J. (eds) Advances in Knowledge Discovery and Data Mining. PAKDD 2012. Lecture Notes in Computer Science, vol 7301. Springer, Berlin, Heidelberg

[LK+07] J. Lin, E. J. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. Data Min. Knowl. Discov., 15(2):107–144, 2007.

[LL+11] Z. Li, K. C. K. Lee, B. Zheng, W. Lee, D. L. Lee, and X. Wang. IR-tree: An efficient index for geographic document search. IEEE Trans. Knowl. Data Eng., 23(4):585–599, 2011.

[NL+11] M. Naphade, D. Lyons, C. Kohlmann, C. Steinhauser, Smart Water Pilot Study Report, IBM Research, 2011.

[Pa16] T. Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In SOFSEM, pages 63–80, 2016.

[PM02] I. Popivanov and R. J. Miller. Similarity search over time-series data using wavelets. In ICDE, pages 212–221, 2002.

[PS+12] Panda S, Sahu S, Jena P, Chattopadhyay S. Comparing fuzzy-C means and K-means clustering techniques: a comprehensive study. Advances in Computer Science, Engineering & Applications. 2012:451-60.

[RG+11] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In SSTD, pages 205–222, 2011.

[SG14] Sehgal G, Garg DK. Comparison of various clustering algorithms. International Journal of Computer Science and Information Technologies. 2014 Apr;5(3):3074-.

[SK08] J. Shieh and E. J. Keogh. iSAX: indexing and mining terabyte sized time series. In SIGKDD, pages 623–631, 2008.

[SL+12] C. Schwarz, F. Leupold, T. Schubotz, T. Januschowski, H. Plattner, S. I. Center, Rapid Energy Consumption Pattern Detection with In-Memory Technology, Int'l Journal on Advances in Intelligent Systems 5 (3 & 4), 2012.

[SM+10] A. Stupar, S. Michel, R. Schenkel, Rankreduce: Processing k-Nearest Neighbor Queries on Top of MapReduce, Proceedings of LSDS-IR, 2010, pp. 13–18.

[SR+15] G. Song, J. Rochas, F. Huet, F. Magoulès, Solutions for Processing k Nearest Neighbor Joins for Massive Data on MapReduce, in: Proc. of PDP, 2015.

[SS12] Singh N, Singh D. Performance evaluation of k-means and heirarichal clustering in terms of accuracy and running time. IJCSIT) International Journal of Computer Science and Information Technologies. 2012;3(3):4119-21.

[SW13] R. Silipo, P. Winters, Big Data, Smart Energy, and Predictive Analytics: Time Series Prediction of Smart Energy Sata, KNIME.com, 2013.

[Ti95] Ho, Tin Kam. "Random decision forests." Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on. Vol. 1. IEEE, 1995.

[YF00] B. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary Lp norms. In VLDB, pages 385–394, 2000.

[ZI+14] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In SIGMOD, pages 1555–1566, 2014.

[ZL+12] C. Zhang, F. Li, J. Jestes, Efficient Parallel *k*NN joins for Large Data in MapReduce, in: Proc. of EDBT, pp. 38–49, 2012.

JK+14 Jung YG, Kang MS, Heo J. Clustering performance comparison using K-means and expectation maximization algorithms. Biotechnology & Biotechnological Equipment. 2014 Nov 14;28(sup1):S44-8.