# Building a Front End Interface for a Sensor Data Cloud

Ian Rolewicz, Hoyoung Jeung, Michele Catasta,
Zoltan Miklos, and Karl Aberer

Ecole Polytechnique Federale de Lausanne (EPFL)
{ian.rolewicz,hoyoung.jeung,michele.catasta,
zoltan.miklos,karl.aberer}@epfl.ch

**Abstract.** This document introduces the *TimeCloud Front End*, a web-based interface for the *TimeCloud* platform that manages large-scale time series in the cloud. While the Back End is built upon scalable, fault-tolerant distributed systems as Hadoop and HBase and takes novel approaches for faciliating data analysis over massive time series, the Front End was built as a simple and intuitive interface for viewing the data present in the cloud, both with simple *tabular display* and the help of various *visualizations*. In addition, the Front End implements *model-based views* and *data fetch on-demand* for reducing the amount of work performed at the Back End.

**Keywords:** time series, front end, interface, model, visualization

## 1 Introduction

The demand for storing and processing massive time-series data in the cloud grows rapidly as time-series become omnipresent in today's applications. As an example, a wide variety of scientific applications need to analyze large amounts of time-series, thus involving more means for managing the data and the corresponding storage systems. Since the maintenance of such systems isn't the main concern for such applications, they often would prefer to lease safe storage and computing power to keep their data without caring about the maintenance or about the hosting, while being still able to run their analyses in place.

For addressing this demand, *TimeCloud*, a cloud computing platform for massive time-series data, is currently being developed at the LSIR[1]. It will allow users to load their data (or register the source of a data stream) and to run analytical operations on the data within the cloud service. This storage-and-computing platform is tailored for supporting the characteristics of time-series data processing, as data is generally streamed and append-only (i.e., hardly deleted or updated), numerical data analysis is often performed incrementally using sliding window along time, and similarity measure among time-series is an essential but high computational operation.

TimeCloud is established upon a combination of several systems for large-scale data store and analysis, such as Hadoop [2], HBase [3], Hive [4], and GSN

[5], but it also introduces various novel approaches that will significantly boost the performance of large-scale data analysis on distributed time-series.[6]

As a part of the platform, the *TimeCloud Front End* was designed to be a user-friendly interface for monitoring and analyzing the data, which also implements a few optimizations that will lighten the work of the back end system. As features of this Web-based interface, we will present the *tabular display*, the *incremental data fetch*, the *model-based approximations* and a set of *visualizations*. Our focus will be made on both the features of the interface and their implementation, and we will also present performance measures demonstrating the improvements made possible by our optimizations.

## 2 Front End Features

### 2.1 The Sensor Table

The Sensor Table is the starting point of our application and is accessible at any point of time from the header menu. It displays all the available sensors for which data is available on the system. The user is then able to select whichever of those he wants to consult, by simply clicking on the corresponding entry in the table. Despite the fact of redirecting the user to the corresponding data tabular display, this table shows information relative to the sensors, like the owner or the accessibility.

### 2.2 Data Tabular Display

The tabular display, as shown in Fig. 1, is the core of the application. It gives us the actual content of the data sent to the system by the chosen sensor. As for an SQL `SELECT *` query, the entire data is available for consultation. Since we are concerned by time-series data, all the entries are indexed and sorted by their timestamp.

We put on top of the table a menu for interacting with the data, along with some information concerning the table itself. The latter is composed of the sensor name and the precision we are currently using for viewing our data. More about precision is discussed later in this Section.

Since we are dealing with large datasets, the two major goals were to ease the navigation of the user through the data and to lighten the workload on the back end. Thus, three features were designed to those extents. The first one is a filter bar, appearing when the user clicks on the "Filter" entry in the table menu. It helps the user focusing on the data of interest by specifying begin and end timestamps. The table is then updated with the values requested, as for performing a range query.

The second is the *Incremental Data Fetch* checkbox located on top of the table. As we are dealing with large datasets, loading the entire data available for a sensor at once would be an overkill, firstly because the backend will have to serve a considerable amount of data, and secondly because the rendering of

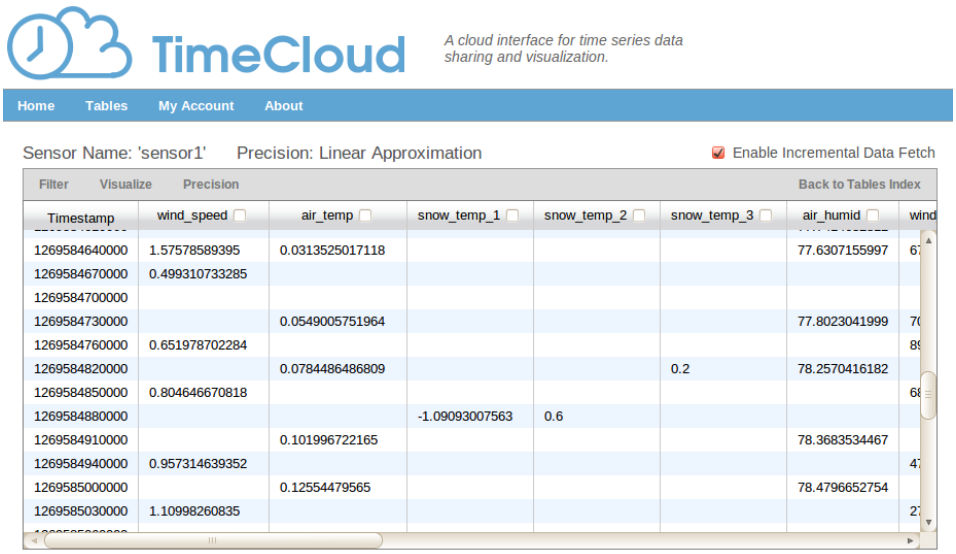


**Fig. 1.** The Data Tabular Display.

the browser on the client side will take a noticeable time to display the whole data. This way, as the incremental fetch is enabled, scrolling to the bottom of the table will fetch additional data asynchronously from the server and append it on the fly at the bottom of the table on the client's browser.

The third feature is the *Model-based Data Approximation*. As we are dealing with sensor data, we expect much of the values sent by the back end to vary not much, which means that we can avoid sending all those values through the network by sending some parameters only and approximate most of the values with the help of mathematical models at the front end. The detailed implementation will be described more precisely under Section 4. For now, we provide two simple models for approximating the data that are selectable from the "Precision" table menu entry, one being the "Adaptive Piecewise Constant Approximation" (being simply renamed as "Constant Approximation") and the other being the "Piecewise Linear Histogram" (being simply renamed as "Linear Approximation"). The precision set by default is the Linear Approximation, but the user can chose at any point of time to convert the content of the table from one precision to another or to get the "Full Precision" data, which is the original data from which the model parameters were computed.

### 2.3 Data Visualizations

Another role for the TimeCloud front end application was to provide to the user a better way of viewing data than with simple tabular display. For viewing precise values, the tabular display is obviously what we need, but for getting a

better understanding of the data as a whole, the precise values reach their limits. This is why, as an additional feature of the front end, we decided to include a set of graphical visualizations for providing more than one way to observe the data.

First, we made the columns of the tabular display selectable, so that the user can chose any columns that he wants to integrate into visualization. Depending if a single column or more than one column were selected, the charts available under the "Visualize" table menu change, since some visualizations were designed for a single variable while others for multiple variables.

Once the column/s is/are selected, clicking on an entry of the "Visualize" menu will display the corresponding chart using the data from the tabular display. As an alternative, the user can also view charts presenting all the columns at once without to have them all selected by clicking on the corresponding entries in the menu.

For now, only a set of basic graphical representations are available, but the underlying system is designed in such a way that it is easily extensible for adding other custom visualizations. Those visualizations are fully computed from the values of the tabular display stored into the JavaScript, which means that we don't query the back end once again for displaying them. Fig. 2 and Fig. 3 show two examples of charts implemented so far.

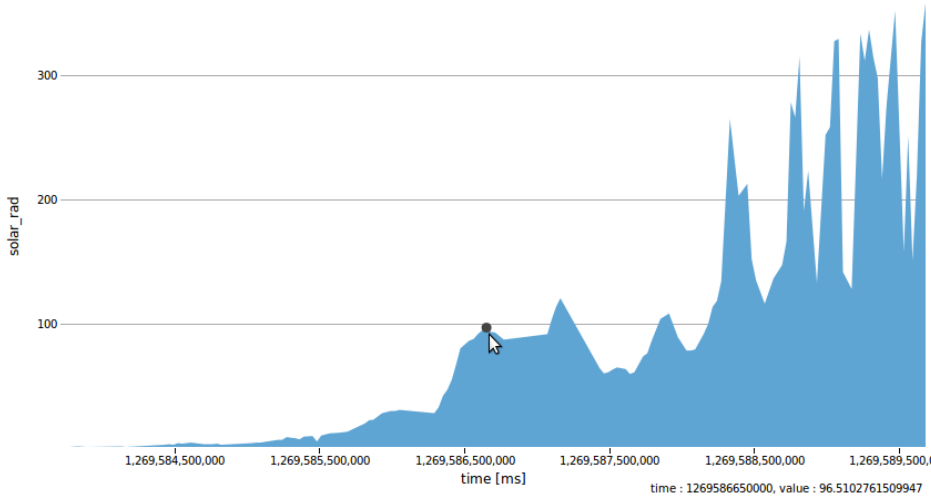


**Fig. 2.** Interactive Area Chart for Single Column Visualization.

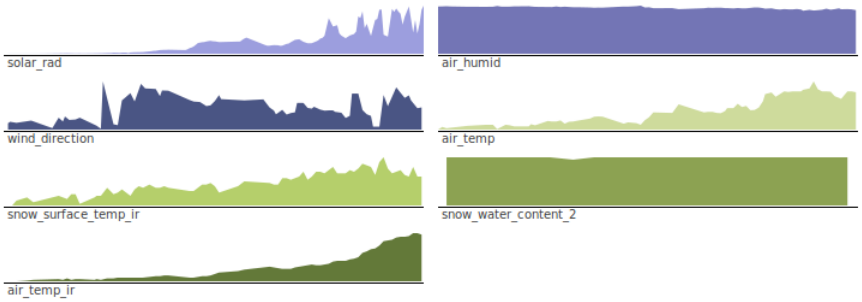


**Fig. 3.** Small Multiples Chart for Multiple Columns Visualization.

## 3 Overview of the Back End

### 3.1 System Overview

Fig. 4 illustrates the architecture of TimeCloud, its Back End consisting of the following major components:

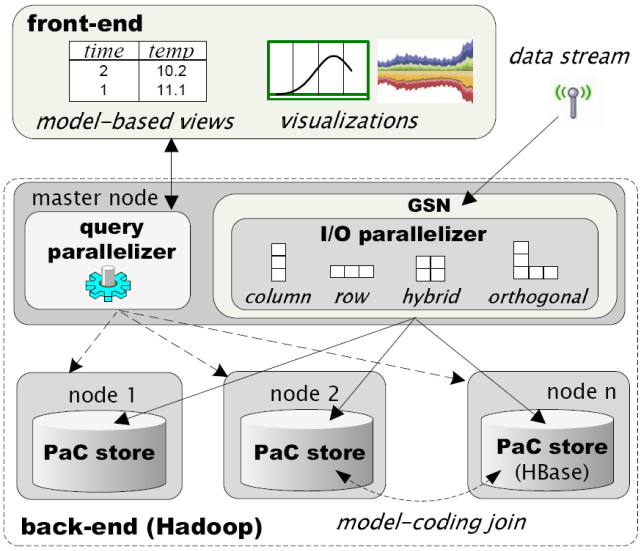


**Fig. 4.** Architecture of TimeCloud.

**GSN (Global Sensor Network)** is a stream processing engine that supports flexible integration of data streams. It has been used in a wide range of domains

due to its flexibility for distributed querying, filtering, and simple configuration. In TimeCloud, GSN serves as wrapper that receives streaming time series from various data sources, e.g., heterogeneous sensors. GSN also allows TimeCloud to execute user-given (pre)processing on raw data on-the-fly, such as calibration of sensor data or adjusting data sampling ratio, before the data is stored in the underlying storage system.

**I/O parallelizer** dynamically distributes the time series streamed via GSN into the back end nodes. At system initialization, TimeCloud applies suitable data partitioning schemes to the dataset initially given. Subsequently, however, new time-series data sources may be registered to TimeCloud (or stored time-series need to be deleted). In these cases, I/O parallelizer computes a new policy how to distribute data. This computation does not begin from scratch, yet incrementally updates the previous results of data partitioning. I/O parallelizer also considers various factors for the data distribution, such as utility ratio of storage capacity and performance statistics on each back-end node.

**PaC store** implements the "partition-and-cluster" store based on HBase. Specifically, when it stores the tuple blocks partitioned and clustered by the I/O parallelizer, it adjusts the physical layout of storage – row, column, hybrid, or orthogonal oriented formats – by exploiting the HBase design idiosyncrasies (e.g. lexicographic order on PK, columnar storage for column families, etc.). TimeCloud then runs queries on the best available data representation dynamically, which is similar to how a query optimizer chooses the best fitting materialized view in data warehouse systems. Simultaneously, the PaC store avoids writing hot-spots that usually plagues range-partitioned distributed storages. In addition, it inherits the features of class distributed storages; each back end node manages a disjoint portion of the whole data while keeping its replicas in different nodes for availability.

**Query parallelizer** balances the back-end nodes' workloads for query processing by monitoring the status of each node for map/reduce jobs. Its key feature is to optimize (time-series) query processing with respect to data partitioning methods configured in TimeCloud. For example, when orthogonal data store is set to the storage scheme in TimeCloud, the query parallelizer runs intra-series queries on the nodes whose data storages are column-stores, whereas it executes inter-series queries on the nodes that have row-stores. This is implemented by extending the HBase coprocessor support.

### 3.2 The Data Model

Since the major component of the back end consists of an HBase instance, the way the data is stored inside HBase becomes of major concern, not only for full precision data but also for the parameters used for model-based approximations.

We are using a single table to store the entries from all the sensors. Each of those entries have a primary key of the form *sensorID:timestamp* which uniquely defines it. The Fig. 5 represents in a schematic way how the data is organized at the back end.

| SensorID: Timestamp | Full Precision | | Linear model | | Constant model | |
|---|---|---|---|---|---|---|
| | temp | wind | temp' | wind' | temp'' | wind'' |
| x1 | v1 | | | | | |
| x2 | | v4 | | | | v13 |
| x3 | | | | | | |
| x4 | v2 | v5 | | (v9, s2) | v11 | |
| x5 | | | | | | |
| x6 | | v6 | | | | |
| x7 | v3 | | (v8, s1) | | v12 | |
| x8 | | v7 | | (v10, s3) | | v14 |
| x9 | | | | | | |

**Fig. 5.** The Data Model.

We create a column family for each of the precisions we are using. In the example, we get a column family for the full precision and two column families for our two model-based approximations. Each of the columns present in the full precision column family will have a corresponding column in the other column families, as we want to have a correspondence between the original sensor data and the parameters that will be used to approximate it.

The full precision data is obviously stored in the system by an external source, so there isn't anything more that we can do about its storage, but the interesting part comes with the storage of the parameters of the models we are using for approximating this data. As we know some common aspects of such parameters, the goal was to store them in a way that was consistent and that reduces as much as possible the amount of redundancy.

The main aspect is that a parameter (or set of parameters) is always linked to an interval {t1, t2} in which we are approximating the actual data by applying a known mathematical function using the parameters and interval bounds. As we know our function and how to apply it, we needed to find a way of making available the parameters and interval bounds without adding unnecessary information in the table. To do so, we used the fact that the set of intervals represents a partition of the whole set of timestamps. Thus, we store the parameters at the end timestamp of the interval they apply to, so that any `GET` query done for some timestamp will return the parameters that apply to the interval the queried timestamp belongs to.

For example, by observing the Fig. 5, the wind parameters for linear approximation stored at index *x8* under the *wind'* column are applicable for the interval {*x5*, *x8*}. We can deduce the upper bound of the interval by the fact that the *wind'* column contains a non-null value for index *x4*, which defines the previous interval. Additionally, due to HBase specificities, querying the *wind'* column at index *x6* will return the first non-null value encountered in or after *x6*, which will be in our case the value stored at index *x8*. The data model described above gives us a nice way of retrieving the data we need for computing our approximated values. The computation of those values is described under Section 4.3.

## 4 Implementation of the Front End

### 4.1 Technologies Used

The Timecloud front end was mainly built using the Python [7] programming language, next to Web formatting and scripting languages like XHTML, CSS and JavaScript. For speeding up the development process, the front end was built with the help of a few framework and libraries.

We first used the Django [8] framework that eases the development of Web Applications following the MVC (Model-View-Controller) software architecture using Python. It comes with a Controller part already coded, which let's you only implement the logic of your application. Additionally, it comes with nice features like a templating engine or predefined routines that also allow extending easily an existing application with new components.

Another library that we used was the YUI 2 Library [9], which consists of a set of JavaScript and CSS tools for easing the task of JavaScript development, as it often is a pain in Web development.

Finally, we used the Protovis [10] library for building our visualizations. This JavaScript library turns parts of JavaScript code into SVG (Scalable Vector Graphics), which offers a variety of possibilities for designing custom charts based on existing data. This library was chosen for its flexibility and its simple setup.

### 4.2 The HBase Interface

As the front end should get the data directly from an HBase instance located at the back end, one of the main concerns was about the interaction between both parties. On one side, we have a large-scale database that comes with an API filled with elementary operations, while on the other side we have an application that needs to perform more complex tasks that involve post-processing of the data retrieved. Because of those concerns, a layer was added between both parties to simplify the data retrieval of the front end and extend the range of possible operations for the back end.

HBase comes with Apache Thrift [11], a software framework for building cross-languages services. Using Thrift, we were then able to generate an API for

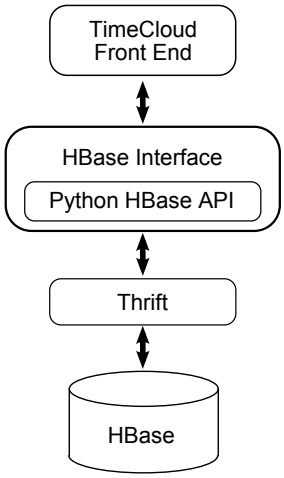


**Fig. 6.** Interaction between the Front End and the HBase instance.

HBase in Python and use it for building our own methods. Fig. 6 shows how the Front End interacts with the HBase instance.

Once the API was generated, we built a set of wrapping methods in Python that we use now for retrieving the data in a format suitable for our needs. Indeed, the API was retrieving and delivering the cells and rows as Objects, which isn't flexible enough, so our methods are retrieving the attributes of those Objects and storing them into Python dictionaries and lists.

Apart from methods used for connecting and disconnecting from the HBase instance, two main methods were developed for retrieving the data from the back end:

```
def extendedScan(self, tableName, prefix, columns, startRow, stopRow)
```

This method is used for retrieving any kind of data in a given table *tableName*. It opens a scanner starting at the given index *startRow* and performs scanner get's until it reaches the index *stopRow* or the end of the table. For each row it scans, it populates a Python dictionary containing the values occurring in the columns having their names in the given column name list *columns*. As *columns* can also be a list of column family names, the method returns, in addition of the values list, a list of column names for which values were encountered. A *prefix* is also to be specified, as the data for all the sensors is present in the same table. This way we avoid retrieving rows for other sensors by giving as a prefix the name of the sensor we are interesting in. We named this method "extendedScan", since a more basic "scan" method was also implemented.

```
def modelScan(self, tableName, prefix, columns, startRow, stopRow)
```

This method is similar to the "extendedScan" method above, but is particularly designed for retrieving the model parameters. As we are "reconstructing"

the values from our parameters and since those are stored at the lower bound timestamps of the intervals they are applied to, we encounter a problem for reconstructing values occurring before the *stopRow* index but after the latest parameters we retrieved. This would mean that a lot of values located at the bottom of our scanning interval won't be reconstructed since their parameters are present after the *stopRow* index. Then, we need to perform the following to get all the parameters:

1. Perform an *extendedScan* from the *startRow* index to the index right before the *stopRow*.
2. Scan the row at the *stopRow* index. Get the set of all column names for which the scan was run. For each of the columns that contain a non-null value for this last scan, remove their name from the column names list.
3. Close the current scanner and reopens a new one using the columns that have their name in the column names list.
4. Scan the next row.
5. Remove from the column names list the names of the columns that had non-null values during the last scan.
6. If there are still any names in the column names list, go to point 3. Else, close the scanner and be done.

This algorithm is illustrated in Fig. 7, assuming we queried for all the values going from index *x1* to index *x5* with linear approximation.

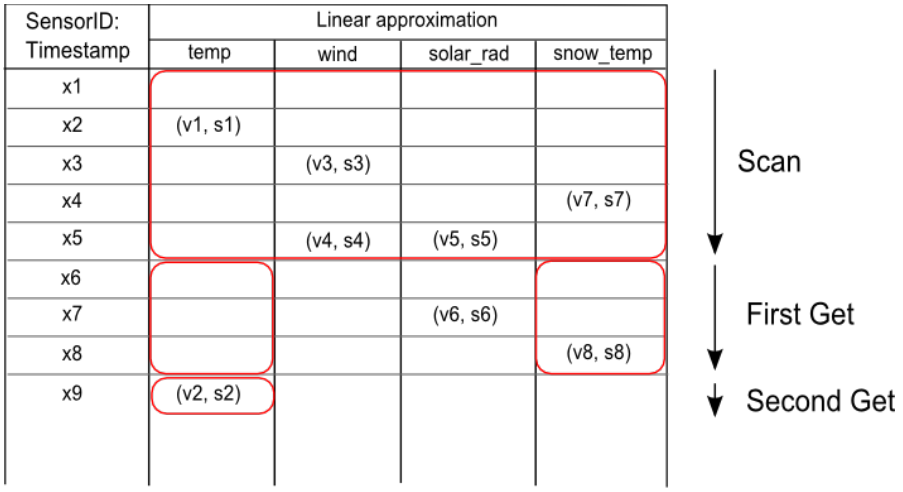


**Fig. 7.** Illustration of a modelscan run.

### 4.3 The Model-based Data Approximations

Before diving into the reconstruction of the data by the front end, we first take a look at the two simple approximation models we implemented for our system.

**Constant Approximation** This model is one of the simplest, as it only approximates a group of contiguous values by their mean. Groups of values are then approximated by a single parameter, and they are in the same interval if their difference $\varepsilon$ with the mean isn't higher than a given threshold. The mean is computed and updated each time a value is appended at a given column. At this moment, we put the new value into the group and a new mean is calculated for it. If one of the values in the group gets a difference $\varepsilon$ greater than the threshold with the newly computed mean, then we remove the new value from the group and the previous mean is written as a parameter into the table at the index of the latest value still in the group. The newly-appended value is then added to a new empty group.

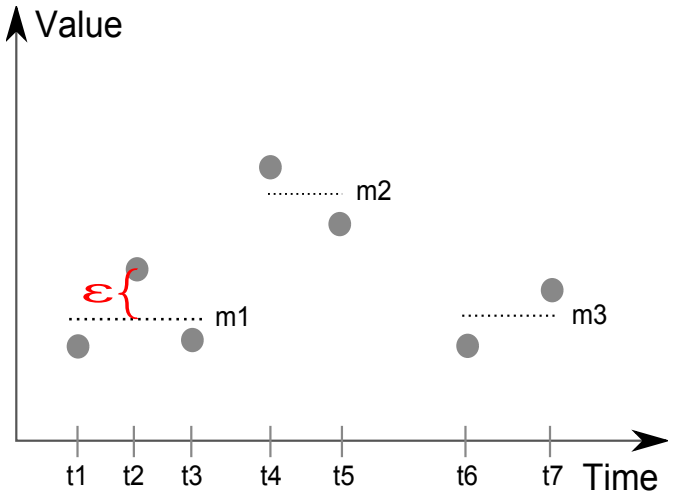


**Fig. 8.** The constant model.

In Fig. 8 above, we can see that the values at timestamps *t1*, *t2* and *t3* will be approximated by the mean *m1*. This mean will be stored as a parameter under the corresponding column in the constant model column family at the timestamp *t3*.

**Linear Approximation** The linear model is very similar to the constant one, but differs in the fact that we are using a linear regression algorithm for computing approximated values. The resulting line can be described by a value at

the origin and a slope, which are the two parameters we are storing in the cells of the linear model column family.

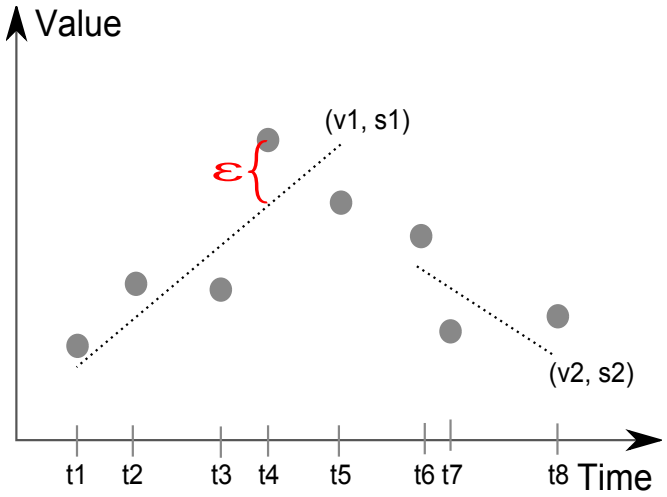


**Fig. 9.** The linear model.

In Fig. 9, we can see that the values going from *t1* to *t5* will be approximated by a line beginning at time 0 at the value *v1* and having a slope *s1*. Those two values will be stored as parameters under the corresponding column in the linear model column family at the timestamp *t5*.

**Data Reconstruction** "Reconstructing" the data out of the parameters only isn't an easy task, and the need of having some global parameters arises when we want to avoid generating values at incoherent timestamps. Indeed, we know that we are approximating sensor data, which means that the sensor should record its values with a fixed time interval between two measures. Knowing this, we should also be able to approximate this behaviour and not only generate approximated values for random timestamps.

Another behaviour observed is that sensors don't necessarily record the values for every parameter they should watch. For instance, the temperature of the snow could be measured less often than the wind speed at a given sensor, so approximating those kinds of behaviours becomes important too, since it prevents us for generating approximated values that don't appear close to an original value.

Knowing this, we need to store this information somewhere in order to retrieve it every time a query for approximated data is received. The thing is that we don't want to include those parameters at the back end as they don't represent a massive amount of data and represent additional queries answered at the

back end. The solution was then to store all this information in a small local sqlite [12] database, which is more than sufficient for the purpose it should serve.

Now that we have access to the parameters linked to the sensor, we have all the needed elements for reconstructing our data. The front end will then follow the procedure below:

1. It queries the back end using a *modelScan* for retrieving the parameters and stores the result of the query locally.
2. It creates a data structure containing a timestamp for each column that it has to approximate values for. Those timestamps will correspond to the latest timestamp at which a parameter was encountered for the column. All the timestamps are initialized to *startRow - 1*. We will refer at this data structure as the "table of latest timestamps".
3. It retrieves from the local database the recording time interval of the sensor.
4. It retrieves from the local database the "steps" data structure of the given sensor. It keeps track, for each column, of the average time interval between two non-null values.
5. It initializes a data structure that will serve as a resulting data structure for the reconstructed data.
6. It looks at the first row of the *modelscan* result. It notes the current timestamp.
7. For each of the columns that have a non-null value, it retrieves the parameters.
8. For each of those columns it gets the corresponding "step", makes sure it is a multiple of the sensor recording time interval and populates the resulting data structure using the parameters. It does so by computing the values backwards, beginning at the current timestamp and calculating previous timestamps with the step of the column. For each of those timestamps, it computes the value with the parameters and stores it into the resulting data structure. It does it only if the timestamps are lower or equal than the *stopRow*. Once the timestamps are lower or equal to the corresponding timestamp found in the table of latest timestamps, it stops generating values.
9. It updates the table of latest timestamps by setting the latest timestamp of all columns that had a parameter with the current timestamp.
10. If there are still rows remaining, it retrieves the next one, notes its timestamp as being the current timestamp and goes to step 7. If not, it ends.

Once this procedure is done, we obtain the same data structure as the one we should expect when retrieving full precision data. The thing is that the values were generated with the procedure and not retrieved directly from the back end.

## 5 Performance Measurements

To measure the advantages of employing Model-based data approximation, we built a testbed on a cluster of 13 Amazon EC2 servers. Each server has the following specifics:

- 15 GB memory
- 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each)
- 1,7TB storage
- 64-bit platform

One server runs the HBase master and the TimeCloud front end (running on Apache2.2 with mod_python). The other 12 servers hosted the HBase region servers (i.e. the servers actually storing and serving the data). Given the variety of data that can be generated by sensors, we synthetically generated a dataset that should represent the worst-case for TimeCloud: a time-series that, when approximated with our linear model, does not compress more than 1/5 of the original dataset, retaining an error $\varepsilon$ of no more than 1 (e.g. highly variable temperature reading). Given a sampling period of 1 second, we also made sure that each interval represented by the linear regression algorithms lasted no more than 5 seconds (by simply generating data in the proper way). Our raw dataset accounted for 100GB, which grew to about 500GB when stored in HBase. The linear approximated dataset size, consequently, was about 28GB – i.e. 1/4 of the original, considering that the representation of an interval is more verbose than a single value. Thanks to some specific characteristics of our back end combined with the design choices we made on the front end, we found some interesting novelties that deserved dedicated benchmarks.

### 5.1 Random Reads

In this benchmark, we show how the linear approximation is useful also in the context of simple random reads (i.e. temperature at a certain timestamp). For 1000 random reads (evenly spread), in the approximated dataset the average performance is a 22% improvement in query execution time. To explain this behaviour, it's important to notice that HBase employs aggressive caching – a smaller dataset will fit more easily in cache. At the same time, the HBase storage files use a simple sparse index to retrieve the given value (by means of consecutive binary searches). Not surprisingly, when a file belonging to the approximated dataset is read (and materialized in the OS caches) it will be able to serve more requests than the original dataset files (thus avoiding expensive I/O operations).

### 5.2 Scans

Scan is the fundamental back end operation needed to plot graphs on the front end. The two datasets were loaded in different column families, such that HBase would store the values in different physical and logical files. Interesting as well, NULL values are not actually stored (differently from widely-available RDBMSs). Such columnar design fits perfectly with our needs, and improves consistently the plotting time of graphs that span over long periods (i.e. the more data it has to be retrieved from the back end, the longer it will take to plot the graph). We show in Fig. 10 the plotting times related to different interval

sizes. It is worth to notice that, for small intervals, the query execution time is dominated only by HBase RPC overhead.

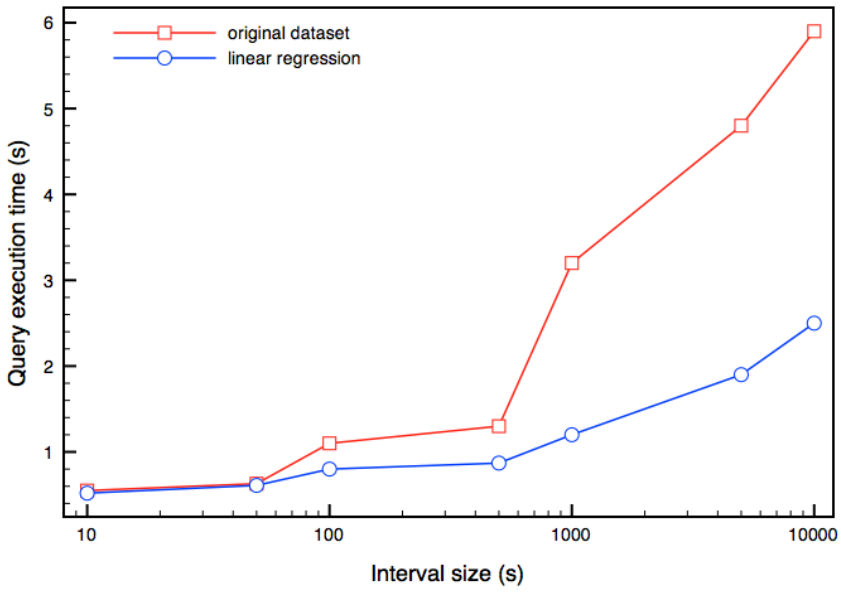


**Fig. 10.** Comparison between original and linearly approximated data for scan queries on different interval sizes.

### 5.3 Network Traffic

Although network traffic between front end and back end depends on many factors (e.g. Thrift optimizations), it was worth to get a ballpark figure of the potential savings we got from the linear approximation algorithm. We then generated a few graphs to initialize the connections to all the slave servers, and then started to measure how much data was transferred from the back end for each new graphs (both for the approximated and full precision versions). We show the results of our measurements in Table 1.

**Table 1.** Comparison between amounts of data transferred over for the network displaying a series of graphs at the front end with both original and approximated versions of the data.

| Graph Number | KB transferred (original) | KB transferred (approximated) |
|---|---|---|
| 1 | 112.3 | 23.3 |
| 2 | 124.5 | 28.0 |
| 3 | 126.6 | 25.9 |
| 4 | 120.2 | 25.1 |
| 5 | 119.95 | 26.8 |
| 6 | 124.4 | 27.7 |

## References

1. Distributed Information Systems Laboratory of EPFL, `http://lsir.epfl.ch`
2. White, T.: Hadoop : The Definitive Guide. O'Reilly Media, 2009.
3. HBase, `http://hbase.apache.org`
4. Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy R.: Hive – a petabyte scale data warehouse using Hadoop. In: ICDE, pp. 996–1005, 2010.
5. Aberer, K., Hauswirth, M., Salehi, A.: A middleware for fast and flexible sensor network deployment. In: VLDB, pp. 1199–1202, 2006.
6. Catasta, M., Jeung, H., Rolewicz, I., Miklos, Z., Aberer, A.: TimeCloud – A Cloud System for Massive Time Series Management. In: SIGMOD 2011 Demo Paper. Introduction.
7. Python Programming Language, `http://python.org`
8. The Django Project, `http://www.djangoproject.com`
9. Yahoo User Interface Library, `http://developer.yahoo.com/yui/2/`
10. Protovis, `http://vis.standford.edu/protovis`
11. Apache Thrift, `http://thrift.apache.org/`
12. SQLite, `http://www.sqlite.org/`