Beyond Discrete Modeling: A Continuous and Efficient Model for IoT

Assaad Moawad, Thomas Hartmann, Francois Fouquet,
Gregory Nain, Jacques Klein, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT),
University of Luxembourg
Luxembourg
Email: first.last@uni.lu

***Abstract*—Internet of Things applications analyze our past habits through sensor measures to anticipate future trends. To yield accurate predictions, intelligent systems not only rely on single numerical values, but also on structured models aggregated from different sensors. Computation theory, based on the discretization of observable data into timed events, can easily lead to millions of values. Time series and similar database structures can efficiently index the mere data, but quickly reach computation and storage limits when it comes to structuring and processing IoT data. We propose a concept of continuous models that can handle high-volatile IoT data by defining a new type of meta attribute, which represents the continuous nature of IoT data. On top of traditional discrete object-oriented modeling APIs, we enable models to represent very large sequences of sensor values by using mathematical polynomials. We show on various IoT datasets that this significantly improves storage and reasoning efficiency.**

***Index Terms*—IoT, Continuous modeling, Discrete modeling, Polynomial, Extrapolation, Big Data**

I. Introduction

The Internet of Things (IoT) and the omnipresence of sensors is considered to become one of the next disruptive changes in our everyday lives [1]. Intelligent systems will be able to analyze our past habits and to anticipate future trends based on data measured and aggregated from various sensors. Since numerical sensor values alone are usually not enough to reason, intelligent systems often leverage modeling techniques to represent structured data contexts. The models@run.time paradigm has demonstrated its suitability to represent and monitor cyber-physical systems and their contexts [2], [3]. Data analytics [4] and machine learning techniques [5] can analyze these data contexts, *e.g.,* to understand and forecast our habits by computing user profiles.

Today's computation theory is based on the discretization of observable data into events associated to a finite time [6]. Following this theory, even the eternity (*i.e.,* long term tasks) can be observed and computed as a sequence of finite temporal data points. Each observable event is then stored by augmenting data with a timestamp, for instance by creating a model element instance with an attribute timestamp [7]. The discretization process of a sensor can easily lead to millions of values in a short amount of time, *e.g.,* considering domains like industrial automation, automated smart homes, traffic monitoring, monitoring of weather conditions, or medical device monitoring. The raise of IoT leads to a massive amount of data, which must be stored and processed. For this reason it is often associated with Cloud and Big Data technologies [8] ,[7]. Specific data structures like time-series [9] [10] can optimize the indexation of mere data. However, the required storage and computation power to store and process this amount of data are a challenge, especially for stream and real-time analyses [11]. Jacobs [12] even calls the modeling of timed data as enumerations of discrete timed values, *Big Data pathology* or *Big Data trap*. Regardless of the chosen solution, intelligent systems need a structured representation of timed data in order to (periodically) reason about it. Therefore, building this context in a way that reasoning processes can efficiently process it, is critical.

Nevertheless, the physical time is continuous and this data splitting is only due to the clock-based conception of computers. In contrary, IoT related data are collected value by value (due to regularly sampled measurements) but are virtually continuous, meaning that, for example for a temperature sensor, at any point in time it should be possible to extrapolate a value (even if there is no measurement at this specific point in time). Indeed, the absence of a data record for a temperature value at a time t does not, of course, mean that there is no temperature outside at this time t . It could simply be that the sensor is saving energy by reducing its sampling rate or it is just not fast enough to measure the temperature at the requested granularity.

The last measured temperature, recorded for time $t-1$, would be the closest value to the requested one. This is due to the fact that the physical value in reality is continuous and with this semantic (taking the last measured value), this can be a solution. In addition, measurement inaccuracies are an inherent part of all physical measurements and strongly considered in signal processing techniques. However, neither the continuous nature of physical measurements nor measurement inaccuracies are considered in modeling techniques. We claim that the traditional mapping between discrete data and model elements is inefficient for the very volatile and continuous nature of IoT related data. In fact, a discrete modeling strategy relies

978-1-4673-6908-4/15 © 2015 IEEE

on the enumeration capability of data that can be, in practice, very costly or even impossible, facing the amount of IoT data. Looking at techniques used in signal processing [13], they offer strategies to represent signals as mathematical functions. In this paper we formulate the hypothesis that techniques from signal processing can create reasoning models that can bridge the gap between continuous volatile data and complex structures.

In a previous work [14], we introduced a modeling technique, which natively integrates versioning on a model element level. This allows to model a context in which every element can evolve independently. It, eventually, enables to *traverse* a model for a particular version whereas in reality it extrapolates the *closest* to the requested version for each element. This allows to simulate the time-continuity of data and we demonstrated the suitability of this approach to handle massive data for reasoning purposes at runtime. However, this previous technique still suffer from the enumeration capability assumption of discrete events, incompatible with the volume of IoT and creates a discontinuity on the measured value.

Because models should be *simpler, safer and cheaper* [15] than the reality, in this paper we introduce a new meta attribute type for models and models@run.time, which can dynamically encode continuous IoT related data. We offer as well an algorithm that enables models@run.time to virtually represent large sequences of sensor values without the need to store all discrete values of the sequences. This keeps the simplicity and expressibility of discrete object-oriented modeling APIs, while using mathematical polynomials to store and read values in an efficient manner. Our approach leverages a live learning strategy, which enables to encode volatile attributes of models and bridges the gap between modeling and signal processing techniques. We integrated our approach into the Kevoree Modeling Framework and evaluated the performance of read, write, and extract operations on IoT datasets from various domains and storage databases.

This paper is organized as follows. In Section II we introduce the foundations of this work: time-series, the associated machine learning for efficient manipulation, and the independent model elements versioning. Section III presents our contribution of continuous models@run.time for IoT domain that we evaluate in terms of performance in Section IV. Then, similar approaches are discussed in Section V and we discuss shortly in Section VI the goal of models versus database. Finally the paper concludes in Section VII.

II. Data Encoding and Modeling Principles

In this section we present concepts from several disciplines required for this contribution.

A. Time Series and Signal Segmentation Algorithms

Time series are one of the oldest [9] structures to represent timed data and even nowadays still one of the most used [16] ones, especially for monitoring metrics. As with most software engineering problems, data representation is a key for efficient and effective solutions [13]. The simplest representation of a time series is a collection of discrete records, each containing a timestamp and the measured value. Such representations are still widely used, because they match perfectly with unstructured storage systems like NoSQL, as for example depicted in the Apache Cassandra blog [7].

The major advantage of this representation is that it does not rely on any knowledge about the continuous nature of the encoded signal. However it is not a very efficient solution. For instance, a constant temperature in a room would be encoded with a number of redundant values, based on the sensor sampling rate, whereas a single value would be sufficient. Even by considering theoretically infinite storage capabilities, like in Cloud computing, reading a large amount of records from disk would be a bottleneck for applications requesting historical values (*e.g.,* to learn patterns or make predictions).

Several higher-level representations of time series have been proposed, including Fourier Transforms [17], Wavelets [18], and Symbolic Mappings [19]. Each of them offer a different trade-off between the compression of the representation and the introduced inaccuracy compared to the original signal. In a nutshell, these techniques rely on a segmentation approach which can be classified as follows [13]:

1) Given a time series T, produce the best representation using only K segments.
2) Given a time series T, produce the best representation such that the maximum error for any segment does not exceed some user-specified threshold, $maxerror$.
3) Given a time series T, produce the best representation such that the combined error of all segments is less than some user-specified threshold, $totalmaxerror$.

Most of these techniques work only in batch mode, thus they require to load and keep all the signal samples in order to compute the mathematical model. This mathematical model can be later used to extrapolate any signal value between the first and the last well-known measures. As argued by Kristan *et al.,* [20], batch processing is incompatible with live reasoning over IoT data and thus also with the envisaged models@run.time paradigm. Firstly, because sensor data are pushed in a stream-based way and thus recomputing the whole model would be very expensive (recomputing for every new value). Secondly, batch processing relies on the enumeration capabilities, which has been depicted as difficult, especially for near real-time reasoning on IoT data. Therefore, for the envisaged models@run.time usage, only techniques which can compress in live *i.e.,* compute signal segments without past values, are appropriate. Using live encoding algorithms, once a signal sample has been encoded into a compressed representation, it can be discarded, therefore such techniques allow to go beyond enumeration limits of samples.

Among the three segmentation strategies, the first and third rely on a strong knowledge of the measured signal. However, such a requirement is usually incompatible with IoT and models@run.time usage. Firstly, models@run.time leverage static information defined in meta models, in which it is difficult to express characteristics about dynamic signal behavior. Secondly, the sampling rate in IoT domain varies

considerably, because sensors often slow down sampling rate for energy efficiency reasons. Thus, knowledge about the best segmentation strategy could be outdated over time, *i.e.,* no longer appropriate. Therefore, the second approach of guaranteeing a maximum error per segment, is the most appropriate, because it is typically domain dependent and it could be defined in a meta model. By using the sensor physical measurement error to guide the compression, we are searching for the most efficient representation for sensor time series according to what we measure from the real world.

To sum up the requirements necessary to be integrated into the models@run.time approach, we need a time series compression and segmentation algorithm which is able to compute mathematical models in live and should be driven by a maximum tolerated error.

One of the most famous and frequently used segmentation solutions in the literature is the Piecewise Linear Representation (PLR) [13]. Intuitively, PLR refers to the approximation of a time series T, of length n, with K straight lines. Because k is typically much smaller than n, this representation makes the storage, transmission, and computation of data more efficient with a compression ratio of n/K. In our work we extended the PLR representation to a polynomial representation, and most importantly, we do the calculations in live (not in batch). We encode the time series into K segments of polynomials of any degrees (instead of lines).

Fitting a polynomial regression of a degree n on a dataset, is a classical mathematical and machine-learning problem. For a given dataset of m pairs $\{x_i, y_i\}$, with $0 \leq i \leq m$, the question is to find the coefficients $\{a_0, ..., a_n\}$ of the polynomial $f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$, that represents the best, with the least possible error, the given dataset. The error minimized is usually the sum of the squares of the differences between the polynomial representation and the actual values y_i in the dataset: $e = \sum_{i=0}^{m}(f(x_i) - y_i)^2$. Many techniques have been proposed and implemented to solve this problem. Most notably, using linear algebra and matrices manipulation, like the EJML linear solver [21]. However these techniques are not adapted for online and live learning, since they require access to past values and that a user specifies the maximum degree of the polynomial. To workaround this problem we leverage a live splitting of polynomials into a sequence of polynomials valid for time boundaries, details are given in the contribution section of this paper. We argue that this representation is efficient in terms of storage and yet very fast to calculate in an online manner, making it a perfect choice to be integrated in a models@run.time approach.

B. Native Independent Versioning for Context Elements

Data composing the reasoning context of intelligent systems can have very different update rates. For instance, a topology related information can be updated every month whereas a sensor related data will be updated every millisecond. The development of such systems, driven by models@run.time, implies an efficient strategy to manage these very volatile data together in a context model with static data. Indeed, saving the whole model for each modification offers a very poor performance, whereas creating timestamped elements for each change also leads to large and unmanageable models.

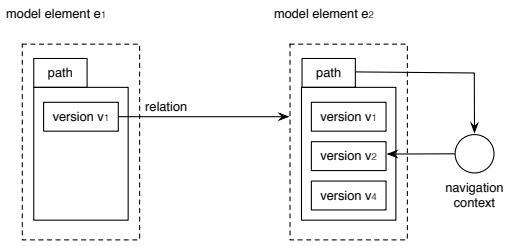

Fig. 1. Native versioning for model elements

To tackle these problems, we proposed in this conference last year [14], a native versioning concept for each model elements. In this past contribution, we defined that every model elements can exist in a sequence of versions, attached to its unique identifier *(e.g., path)*. We also defined model element relationships as links between unique identifiers, without the version information. We presented as well a novel navigation concept that always resolves the *closest* version to the one requested. For instance, if the requested version is $v3$, the navigation context will resolve the closest one $v2$ in case $v3$ does not exist. This is illustrated in Figure 1. each attribute change, a new version is created with the time as the version ID, our previous work enables to model classical discrete time-series. The time-continuity is simulated by the navigation context resolver which takes the closest version of the requested one. For example, if the sensor has pushed values at time t and $t + 100$, any get request on the attribute between t and $t+100$, let's say at $t+50$, will resolve the version stored at time t. However, this creates a discontinuity on the attribute-value dimension. A linear signal for example will be transformed to steps-shaped signal. We would expect that the value change smoothly from time t to $t + 100$. Moreover, based on this discrete representation of versions, our previous approach still expects the enumeration capabilities of versions and thus is limited to face a high sampling rate from a sensor.

III. A Continuous and Efficient Model for IoT

Models@run.time and $models$ in general are defined by the aggregation of a set of model objects. Each $object$ contains a set of $attributes$ and a set of $relationships$. In our previous work, and in order to enable independent model elements versioning [14]. This definition is based on a discrete strategy which defines a new segment or version in case of modification of any attribute of the object. As explained in the previous section, while this strategy succeeds to imitate the continuity through segment transparent resolution, it will lead to the creation of one segment and one entry in the index tree for each variation of any attribute. Therefore, in case of several attributes change into the model this solution is sub-optimal.

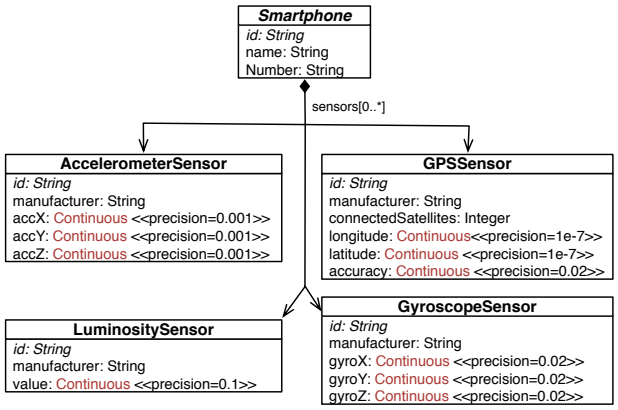

Fig. 2. A smartphone meta model

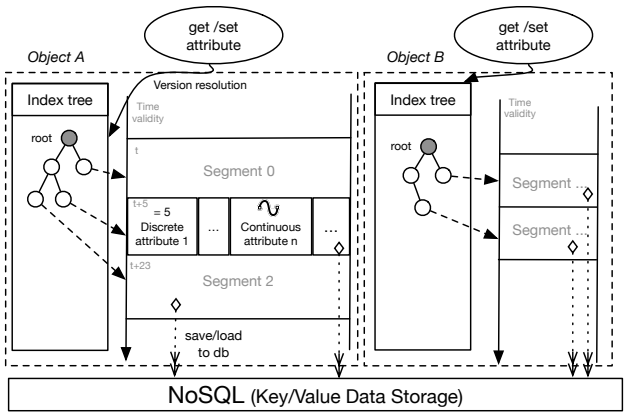

Fig. 3. Structure for Models with Discrete and Continuous Attributes

A. Continuous Models Structure

In this paper, we introduce a new meta-attribute named **continuous**. Figure 2 shows the usage of such type in a meta model of a smartphone. A smartphone has a collection of sensors, each sensor has several attributes. Some of these attributes represent physical continuous signals and they are marked as continuous in the meta model. The precision depicts the maximum tolerated error we can use for our representation and segmentation. In the previous work, all of these attributes are encoded as discrete doubles, and each minor change of any value of any of these attributes will create a new version of the model. Our hypothesis is that, similarly to time-series compression techniques, a model could detect the relationship between consecutive write on numerical values and try to encode them in an efficient manner. This way, the model will transparently reduce the number of created segments per object while allowing a reasoning engine and the sensor to perform classical get and set operations on any model element attribute.

However, models should be able to contain non continuous attributes such as a string or a reference, any modifications of these will lead to the creation of a new segment as well. For example, in our meta model example, the number of connected satellites in the GPS class is an integer and is not continuous. Each modification on this attribute (aka number of connected satellites change) will create a new segment, regardless if the other attributes (latitude, longitude) can be still encoded in the same continuous polynomial function.

In the Figure 3 we depict this new modeling architecture which is able to manage seamlessly continuous and discrete classical attributes *(e.g. string, boolean...)*. Each object has a balanced tree structure to index the segments it contains. Segments are considered as the smallest data granularity, which can be saved and loaded from scalable storage like NoSQL or Distributed Hash Tables technologies. Each segment contains the payload of a version for a model object. This payload contains discrete information like integer values, or functions which can encode several values in a time-series manner of a continuous attribute. Each time a get operation is performed on the model on a continuous attribute, the corresponding segment is loaded and the attribute value is extrapolated using the function stored into the segment. Conversely, each time a set operation is performed on a continuous attribute the model has to decide whether to store it in the existing function (update the function) or whether to create a new one in a new segment for this update. As explained in the data encoding principle section, in this paper we have defined an online learning algorithm that can build this series of segments, minimize the total number of segments needed while ensuring a maximum tolerated error for the created functions according to the precision of the attributes defined in the meta model. The next subsection details this segmentation strategy, the following one shows the construction of the segments and their continuous functions based on polynomial representations.

B. Live Model Segmentation Driven by Tolerated Error

A main target of our approach is to produce as few segments as possible under the following design constraints:

- **Computation requirement**: Each segment should be built in a fast online processing way (providing a fast setter) and should be able to compute fast the attribute whenever asked (providing a fast getter).
- **Error requirement**: All continuous attributes in a segment should not deviate from the original data more than the specific tolerated error defined in the meta model.
- **Compression requirement**: The model element will not create a new segment as long as, in its last segment version, the error requirement is met in all the continuous attributes and as long as the other non-continuous attributes did not change. This is to minimize the number of segments created, and to keep the model size minimal.
- **Time validity**: Each segment is valid from its creation time *(time origin)* up till the time origin of the following segment. The time origin of each segment is used as the reference starting point for the attribute functions.
- **Continuity requirement**: The first value of a continuous attribute calculated by one segment should be the same as the last value generated by the previous segment.

- **Self-containment requirement**: Each segment should contain all the information needed for its own process (compression, reconstruction).

Formally, in our current implementation, a **model element** contains a unique ID and an Index tree $tree$.
The **index tree** is a sequence of segments: $tree = \{s_1, \ldots, s_i, \ldots, s_n\}$.
A **segment** s_i is a 4-uple $s_i =< t_{oi}, C_{Ai}, D_{Ai}, R_i >$, where t_{oi} is a time origin, C_{Ai} is a set of continuous attributes, D_{Ai} is a set of discrete attributes, and R_i is a set of relationships.
A **continuous attribute** $c_{ij} \in C_{Ai}$ is a sequence of weights of the function that represents it. $c_{ij} = \{\ldots, w_{ijk}, \ldots\}$. In our implementation, we use the polynomial function:
$f_{ij}(t) = w_{ij0} + w_{ij1}(t - t_{oi}) + ... + w_{ijn}(t - t_{oi})^n$,
as the retrieval function to get the value of a continuous attribute at a time t, with $t_{oi} \leq t < t_{o(i+1)}$ (time validity requirement). The polynomial choice is to meet the computation requirement, it is efficient to build in live using EJML [21]. Moreover, from the error requirement, we derive the following continuous segmentation condition: $\forall j, |f_{c_{ij}}(t) - y_{c_{ij}}(t)| < \epsilon_{c_{ij}}$. where $y_{c_ij}(t)$ is the physical measured value of the attribute c_{ij} at time t, and $\epsilon_{c_{ij}}$ the maximum tolerated error of this attribute as defined in the meta model.

From the continuity requirement, we get the initial condition of the polynomial functions when starting a new segment: $\forall i, \forall j, f_{ij}(t_{oi}) = f_{(i-1)j}(t_{oi})$. This leads to the following fact: $\forall i, \forall j, w_{ij0} = f_{(i-1)j}(t_{oi})$. Meaning that the constant part of the polynomial is always equal to the latest value generated by the polynomial of the previous segment, for all segments i and for all continuous attributes j.

The self-containment requirement can be easily checked by the fact that the polynomial uses only the weights w_{ijk} and the time origin t_{oi} in its calculation. All this information is stored within the segment.

Finally, each segment can be serialized and de-serialized. It forms the payload to be stored/retrieved in/from a database as presented in figure 3. Note that, our design to encode all attributes in the same segment is to reduce the total number of segments created. This way, if many attributes change at the same time, all of the changes will be encoded in one segment rather than in many segments, to speed up the lookup on the index tree. Whenever a get or set operation is done on a model element, the index tree structure finds the closest segment to the requested time. This has a complexity of $O(log|S|)$ where $|S|$ is the number of segments present in our model. Then the segment delegates the get/set operation to our algorithm that we present in the next section.

C. Online Segment Construction Based on Live Machine Learning

In this section, we present the algorithm that takes care of deciding the degree of the polynomials, calculating their weights, segmenting and storing them whenever the previously discussed requirements are not met. All this should be done in an on-line live learning manner, completely transparent for the final users and while guaranteeing the maximum tolerated error ϵ for each continuous attribute. Figure 4 shows an example of a segmentation of a continuous attribute with the real points that were used to build the segments and the different polynomial representations.

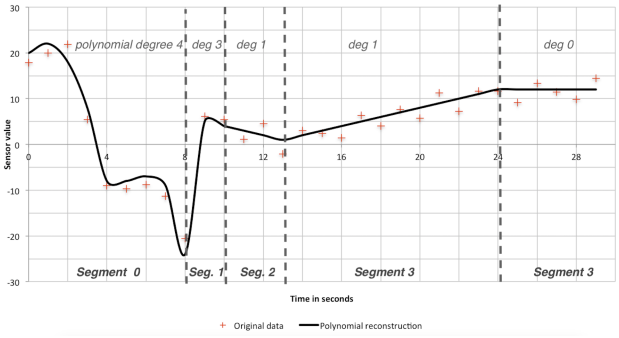

Fig. 4. A segmentation and polynomial encoding of a continuous attribute

The main functionality of our algorithm is in the insert function presented in listing 1. In a nutshell, when the first sensor sample is fed to our model, a polynomial of degree 0 is created from this sample. As long as new values fit in the current polynomial function within the maximum tolerated error for the current degree, the current polynomial model is maintained. When a value does not fit in the current model, we check first if we can increase the polynomial degree, i.e., if (degree+1) is lower than the maximum degree *MAXDEG*. If this is the case, we regenerate a temporary set of samples from the current polynomial, and add the newly arriving value to this set, and we try to reconstruct a new polynomial of a higher degree (degree +1) using EJML linear solver [21]. We send to this linear solver the live generated dataset, and the desired polynomial degree, it returns back the coefficients of the polynomial that fits the best the dataset.

We then do a check to test if the new polynomial represents the generated dataset within a certain tolerated error which depends on the maximum tolerated error specified by the user and the current degree of the polynomial. If this is the case, we update the polynomial to the new representation. If not, this means that increasing the degree will not guarantee that the representation fit within the maximum tolerated error specified by the user, or the polynomial has reached the $MAXDEG$. So the new sample does not fit in the previous polynomial and we were unable to increase its degree, thus the only solution is to create a new segment. In this case, the previous polynomial function is stored, and a new polynomial function is created from the new sample which could not fit previously, and the process restarts.

Listing 1. Our implementation

```
void insert(long time, double value) {
  /*if null init a polynomial of degree 0*/
  if (weights == null) {
    deg=0;weights = new double[1];
    initTime=time; weights[0]= value;
  }
  /*test if the sample fits in previous function*/
  if (Math.abs(get(time) - value) <= maxError(deg)){
    return;
```

```
  }
  /*if not, increase previous function degree*/
  do {
    deg++;
    double[] times = new double[deg+1];
    double[] values = new double[deg+1];
    /*Regenerate values from previous polynomial*/
    for(int i=0; i<deg; i++){
      times[i]= initTime + (i/deg)*(time-initTime);
      values[i]= get(times[i]);
    }
    /*add the new sample*/
    times[i] = time; values[i] = value;
    /*fit a new polynomial of a higher degree*/
    pf = new PolynomialFitEjml(deg);
    pf.fit(times, values);
    //test if the polynomial fits within maxError
    if(maxError(pf.getCoef())<=maxError(deg)){
         /*if yes, we keep this new function*/
         weights = pf.getCoef();
         return;
    }
  } while(deg<MAXDEG);
  /*if degree doesn't fit within MAXDEG,*/
  /*a new segment will be created*/
  segments.save(initTime, weights);
  /*reset the process*/
  weights=null; insert(time,value);
}
```

Our model guarantees that the polynomial representation does not deviate from the real value more than a maximum tolerated error ϵ provided by the user. However, because we are constructing the polynomial on the fly, starting from a degree 0 and upgrading the degree step by step, the error cumulates. Thus we need to define a maximum tolerated error strategy $e(d)$ for each step of the construction (aka. when the polynomial is of a degree d), in a way that the total cumulated error is ensured not to exceed the maximum tolerated error defined by the user. In other words: $\sum(e(d)) \leq \epsilon$.

Another requirement is that the more complex the polynomial representation becomes, the less deviation from the previous representation, we will be ready to tolerate. Otherwise, the total cumulated error will not converge. In mathematical terms: $e(d+1) < e(d)$.

Combining these 2 requirements, our design choice is the following: $e(d) = \epsilon/(2^{d+1})$. One can easily verify both properties: First it is a constantly decreasing function, and second the sum of the error will be always less or equal than ϵ because $\sum(1/(2^{d+1})) = 1$.

In this way, when the polynomial representation is a constant (degree 0), the model can deviate maximum of $\pm\epsilon/2$ from the real values. When we increase the degree to 1 (a line), we can tolerate $\pm\epsilon/4$ more error deviation. The cumulated error for this step is less than $\pm 3\epsilon/4$. And so forth, if the degree of the polynomial reaches infinity, the maximum deviation between our model and the reality will be $\pm\epsilon$, the one specified by the user in the meta model.

IV. Experimental Evaluation

The main goal of our contribution is to offer an efficient models@run.time structure to deal with IoT related data. Thus, this evaluation section puts the emphasis on the performance of our modeling structure and its ability to imitate and rebuild the initial continuity of IoT data. As key performance indicator *(KPI)* we selected classical **write**, **sequential read**, **random read** operations and **storage size**. In addition, we introduce the KPI **continuity reconstruction ability *(CRA)***, which aims to evaluate the ability of the modeling layer to rebuild a continuous signal – even in case of sampling rate variation of one model element. To conduct these experiments, we reused our past work [14], which has been demonstrated to be highly scalable following a discrete strategy. We enhanced this implementation with the continuous polynomial based strategy, and evaluated it on various IoT data sets, with and without this new optimization. The rest of this section is organized following the above defined KPIs, and in addition we add a subsection to show that these results are valid, regardless of the chosen data store implementation *(Google LevelDB or MongoDB)*, up to a common multiplicative factor.

A. Experimental Setup

We integrated our optimization approach into the Kevoree Modeling Framework [1] *(KMF)* and all experiments have been conducted on the latest java version 8. The implementation of our approach is available as open source [2]. We implemented two live learning segmentation strategies. First, the with Piecewise Linear Representation *(PLR)*, which is, to the best of our knowledge, the most efficient implementation for time-series compression, and second with our polynomial extended version. The experiments for all KPIs have been carried out with KMF using Google's LevelDB [3] as a database. To connect KMF to LevelDB we implemented a corresponding driver. The only exception of this is the database impact benchmark at the end of this section, where we use in comparision MongoDB [4] as a database. Again, in order to connect KMF to MongoDB we implemented a corresponding driver. In addition, for all experiments we used the models@run.time in a stream mode, meaning that values are propagated as soon as available. A batch strategy could offer a better performance for raw inserting of values but would introduce a delay, which is in many IoT systems unacceptable. Nonetheless, in the last section of this evaluation we present results for both batch and stream modes, to demonstrate that our approach offers the same advantages for batch and stream (excluding a multiplication factor).

As argued before, IoT data have very specific characteristics that make them challenging to store and process. In order to properly represent these characteristics we selected seven different datasets, containing data from the best to the worst case for our approach. All measurements have been carried out on an Arduino platform and all of them have been normalized to **5 000 000** entries. Table I presents a summary of the datasets and their properties:

The first two datasets are artificial and represent the best two case scenarios: a sensor measuring the same value overtime, or a linearly growing measure. The third and fourth datasets are measured from temperature and luminosity sensors, like

[1] http://kevoree.org/kmf/
[2] https://github.com/dukeboard/kevoree-modeling-framework
[3] http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html
[4] http://www.mongodb.org

Database	Sensor
DS1: Constant	c=42
DS2: Linear function	y=5x
DS3: Temperature	DHT11 (0 50'C +/- 2'C)
DS4: Luminosity	SEN-09088 (10 lux precision)
DS5: Electricity load	from Creos SmartMeters data
DS6: Music file	2 minutes samples from wav file
DS7: Pure random	in [0;100] from random.org

TABLE I
EXPERIMENTAL SELECTED DATASETS

the ones that can be found in personal weather stations. The fifth dataset is issued from our industrial partner Creos Luxembourg S.A. which conducted experiments in three regions in Luxembourg to measure the electric consumption based on smart meters. The sixth dataset is an open source music file, which has been sampled into chunks of amplitudes. Finally, the seventh dataset is a purely randomly generated set of doubles.

The selected datasets highlight three very different types of IoT data. DS1 and DS2 are aperiodic (nearly constant) signals, while DS3, DS4, and DS5 contain periodic signals *(hourly, daily...)* and finally DS6 and DS7 are (almost) random signals.

All experiments have been carried out on a Core i7 computer equipped with 16GB RAM and SSD drive.

B. KPI Write Evaluation

In this first benchmark we evaluate execution time to insert data in a stream mode. Therefore, we use the data from our datasets DS1 to DS7. We run the experiments three times per dataset. The results are depicted in Figure 5.

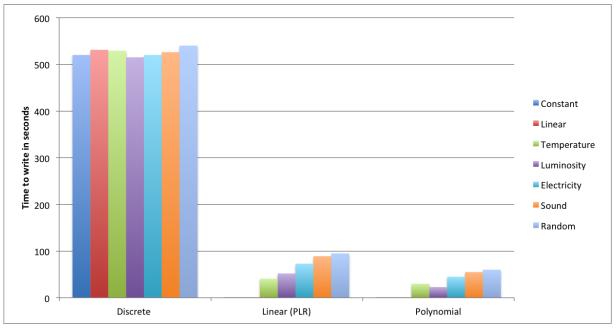

Fig. 5. Time to write on Leveldb for the 3 methods

The evaluation criteria for this KPI is execution time, which is represented on the vertical axis. On the horizontal axis we depict all datasets organized by the segmentation method. As segmentation methods we use discrete, PLR linear, and our polynomial-based live compression method. All values are saved in a database (SSD drive).

The discrete modeling method needs roughly the same time to write for all datasets. This is not a surprise since this time depends only on the number of samples, which has been normalized to 5 000 000 elements for all datasets. The observed small variations can be explained by the different numbers of bytes to represent each value *(mainly depending on the precision of the signal)*. For the linear and polynomial representations we observe very different times for the different datasets. This is because the time depends on the number of generated segments, which itself depends on the dataset. The constant dataset results in the fastest time and the random dataset in the worst time.

However, we observe that in all cases the polynomial is faster to write compared to a complete discrete method. In fact, the polynomial is very close (or even similar) to a PLR method. Through this benchmark we demonstrate that the time spent to compute polynomial representations and compress the signal is still below the time to store all enumerated values, even on a fast SSD drive.

C. KPI Sequential Read Evaluation

In this benchmark we evaluate the time to read a sequence of elements from the model. We select this operation for the KPI sequential read because it is the most used operation in machine learning or pattern-detection algorithms that need to access model elements sequentially to compute a user profile or a similar trend. For instance, in domains such as smart grids, the sequential read is mostly used to extrapolate electric consumption data that feed an alert system. The sequential read is thus performance critical for IoT oriented models@run.time. Again we run the experiments for DS1 to DS7 and use the three live compression methods. Results are depicted in Figure 6.

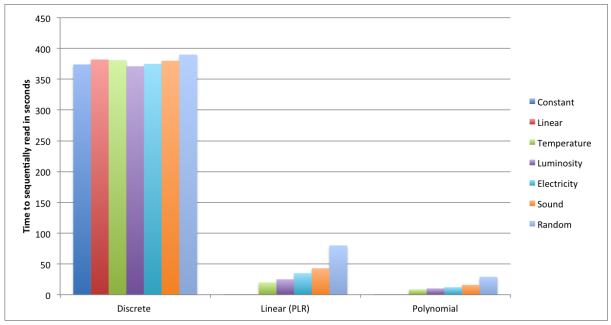

Fig. 6. Time to read sequentially from Leveldb for the 3 methods

Again, the vertical axis represents the time to read all dataset values and the horizontal axis depicts the different datasets and method combinations. Similarly to the write benchmark, the discrete method introduces very few variations in time due to the normalized datasets. However, we can observe a significant improvement factor, between discrete and, PLR or polynomial representations. We can also notice a significant improvement of polynomial versus PLR. The polynomial can read thousand times faster for the best cases: constant or linear dataset. This can be explained by the fact that both, the PLR and polynomial models encode the constant or the linear function in one function. For the other datasets the average speedup in sequential read is between 30-50 times faster. This significant improvement factor can be explained by the fact that a compressed representation can contain many points, reducing both the amount of database calls but also the size of

segment indexes. Moreover, the polynomial capacity is much better than the linear one. This explains why we can observe a speedup between 2-3 times between both approaches.

D. KPI Random Read Evaluation

In this section we conduct benchmarks to evaluate the Random Read KPI. Here we are evaluating the capacity of models@run.time to offer scalable read operations for elements randomly distributed in the model itself. This operation is a good example for applications that need to lookup random elements (ex. Monte-Carlo simulations). Again this operation is widely used in intelligent system and performance is critical. We use the same experimental setup than for the sequential read, except the model element selection simulates a random process. Results are presented in Figure 7.

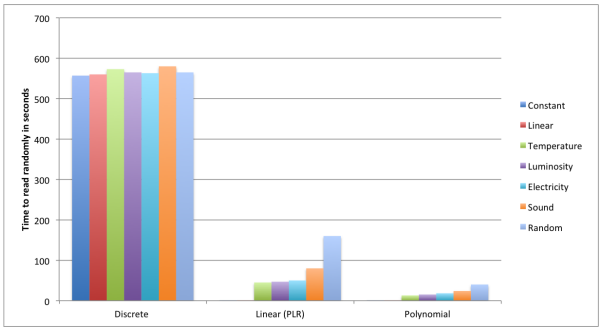

Fig. 7. Time to read randomly from Leveldb for the 3 methods

Again, results are similar to the sequential read, except that the PLR performs worse than for the sequential read. In contrast, the polynomial encoding can still be up to thousand times faster for the constant or linear dataset. For the other datasets, the average speedup is between 40-60 times.

E. KPI Storage Size Evaluation

In this section we conduct experiments to evaluate the gain in terms of storage space. Indeed, both the PLR and polynomial implementations try to reduce the number of segments and thus reduce the overall models@run.time storage requirements. Using the same setup used for *KPI Write*, we observe two things for the *KPI Storage Size*. First of all, the number of segments after live insertion into the models@run.time. This number reflects the pure compression of the signal in memory. Secondly, we measure the amount of bytes, which are sent to the NoSQL storage through the models@run.time layer. This evaluation takes the overhead of models into account and validates the performance in an integrated environment. In Table II we present the segmentation results.

In this result we are considering the 5 000 000 elements inserted into the discrete method. The compression rate is computed according to the size to represent a long value in the Java Virtual Machine. In other words, this compression rate is equivalent to the memory saved by the virtual machine to represent the signal, expressed in bytes. For the polynomial method, on a constant or linear signal the compression is maximal, at around 99%. This compression is still between a range of 73 to 46% for IoT datasets and fall at 33% and 10% for random signals, which are our worst case scenarios.

Database	# Seg. PLR	Rate	# Seg. Poly	Rate
Constant	1	99.99 %	1	99.99 %
Linear fct	1	99.99 %	1	99.99 %
Temperature	316,981	93.66 %	78,580	73.54 %
Luminosity	2,554,227	48.91 %	296,362	63.44 %
Electricity	3,966,521	20.66 %	396,209	46.42 %
Music file	4,508,449	9.83 %	461,603	33.21 %
Random	4,999,824	0.01 %	682,324	10.53 %

TABLE II
NUMBER OF SEGMENTS AFTER LIVE INSERTIONS AND RATE

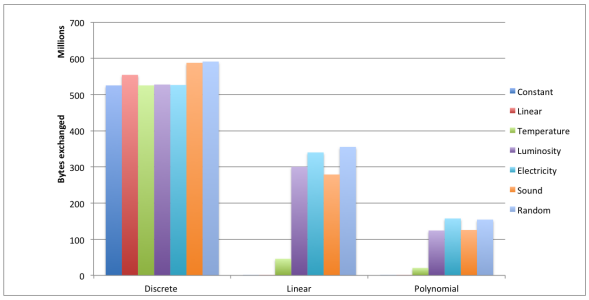

Fig. 8. Bytes exchanged between NoSQL storage and models during 5 000 000 write operations

In this graph (Figure 8) the vertical, respectively horizontal axis represent the bytes stored in the database and the combination of methods and datasets. We observe a similar trend like when we operate in-memory, however, the compression rate, in average, is less efficient. This is due to the overhead of the serialization strategy, which converts long into a byte representation in order to be transferred to the database layer. In fact, polynomials use high precision double numerical values, leading to very long string representations, which makes the compression rate slightly weaker. This overhead introduced by the implementation of KMF can be optimized in future work by using a binary serialization format.

F. KPI Continuity Reconstruction Ability Evaluation

In this experiment, we measure the ability of the model to reconstruct the value-continuity. The goal is to be as close as possible to the original signal in case of samples loss. The motivation behind this experiment is based on sensor network properties. In fact, due to network errors or simply in order to reduce energy consumption, sensors can change their sampling rate. Here, we evaluate the capacity of our continuous model to detect such missing values and extrapolate them using the various segmentation strategies. We engage the same experimental setup than the KPI Write benchmark, except that we uniformly drop an average of one value out of 10. Later, in an experiment similar to the sequentially read, we reconstruct the missing samples and we measure the error *(deviation from the real missing value)*. Our results are presented in the Table III.

Database	Discrete	Linear	Polynomial
DS1: Constant	0%	0%	0%
DS2: Linear function	5 %	0%	0%
DS3: Temperature	8.5%	3%	3%
DS4: Luminosity	9.9%	3.6%	3.5%
DS5: Electricity	17 %	7%	6%
DS6: Sound sensor	21%	15%	13%
DS7: Random	31.8%	31.1%	30.8%

TABLE III
AVERAGE ERROR FOR BOTH STRUCTURES WHEN WE TRY TO APPROXIMATE MISSING VALUES

Each extrapolation introduces a deviation, however through our experiments we highlight that the polynomial strategy compensates best for lost samples. For the random dataset, all methods lead to almost the same recovery error due to the random nature of the dataset. The PLR approach was already better than the discrete strategy, which imitates more or less a square signal. However, in all cases the polynomial method offers a better trade-off and is more resilient to data loss.

G. Results Generalization

As discussed before in the experimental setup subsection, the database used for the experiments and the mode *batch or stream* influence the absolute values for the various conducted benchmarks in a significant manner. In this last benchmark we evaluate experimentally this threat to validity by running the same write operations on various configurations. We run the *KPI Write* benchmark two additional times in *stream* and *batch* mode, once on top of the *LevelDB* driver and the *MongoDB* driver. Results are presented in Table IV.

Test	LevelDB	MongoDB
Stream Discrete	500 to 520s	680 to 750s
Stream Polynomial	1 to 60s	1 to 80s
Batch Discrete	70 to 75s	77 to 84s
Batch Polynomial	1 to 10s	1 to 15s

TABLE IV
TIME RANGE IN SECONDS FOR SAVE IN LEVELDB AND MONGODB

Results are presented in time (seconds) and in range (from best to worst case based on the different datasets). As expected, batch processing offers better performance to insert 5 000 000 model elements and this regardless of the chosen database implementation. Comparing the two databases it is interesting to notice that the performance difference of MongoDB and LevelDB is consistent, at around 33% to 45%. We argue that the chosen database can only impact our results with a multiplication factor. In the same manner the ratio between polynomial and discrete is also consistent between the stream and batch mode, with a factor of roughly 11 to 17. We argue here that the mode only impacts with a multiplication the final results. Overall, we can assess the validity our results if considering the improvement factor and not raw values, regardless of the chosen mode or database.

H. Experimental Results Summary

In this section we have conducted various benchmarks to demonstrate the general application of our polynomial segmentation method for models@run.time optimization of IoT data. The theoretical limit of our polynomial approach is based on the maximal polynomial degree constructed during the segmentation method. During all this experiment we demonstrated that our chosen strategy with a **maximal degree of 20** applies the best for all the evaluated datasets. We also argue that such strategy improves both time and value continuity.

V. RELATED WORK

Polynomial Approximation has been used previously to represent ECG signals in [22]. In their work, the authors were able to achieve high transmission bit rate for these signals and a high accuracy of reconstruction. However, their work is very domain specific, they use the ECG domain-specific knowledge (The ECG period) in order to divide the signal into chunks. Each period of the signal is represented by one polynomial.

In another work, polynomial approximations have been used for speech compression [23]. In our work, we provide a generic modeling framework and algorithm that can operate on any signal from any domain, even if it is non-periodic. The signal division into chunks is mainly driven by the maximum tolerated error which is domain and sensor specific.

Chen et al [24] presented a compression algorithm called Rate Adaptive Compression with Error bound (RACE) that attempts to compress sensor data in order to meet their limited bandwidth and battery source requirements. The algorithm is based on the Haar wavelet transform. This work shows as well the importance of having a good representation for time-series especially when there are power or network constrains. However they process in batch mode of 512 samples. Our approach can work from full live (update the model every sample), till any batch mode size and thus is more suitable for real-time monitoring where reactivity is required.

The BlinkDB project [25] is a query engine targeting has well BigData. Similarly to our approach, BlinkDB leverages a domain tolerated error in order to reduce the volume of loaded data to resolve a MapReduce query. Such approach offer an efficient a posteriori solution to query large datasets of time-series oriented data. However our approach is integrated in the core of data structure, allowing an a-priori improvement of query but also storage.

VI. DISCUSSION: DATA-DRIVEN MODELS OR MODEL-DRIVEN DATA?

While models@run.time and modeling in the large techniques are progressing further and further, the border between databases and models tends to blur more and more. This trend is even more reinforced by recent NoSQL techniques and other storage systems for unstructured data, which leave to other software layers the freedom to define the semantic and the structure of data. This enables data storages to efficiently scale horizontally, as can be seen in many Big Data approaches.

The vanishing border between models and data storage systems inspires this discussion about the role of data storage systems and models. Our contribution reflects this trend since models@run.time becomes a model-driven data structure, supporting the creation of data-driven intelligent systems. This

pushes the management of data partly into the application layer and allows to use simpler storage solutions, like those developed for Big Data storage systems.

Beyond structural information, a model can define a rich semantic, like our definition of continuity for attributes. Leveraging this additional information, a model can organize and process data in a very efficient way, like we suggest with our polynomial approach. With traditional data-schemas a storage can hardly perform such optimizations, because it only stores a stream of values. Although it optimizes the storage, our objective is to offer a rich semantic to data structures, which is one of the primary goals of models.

VII. Conclusion

The Internet of Things and smart systems with their reasoning abilities are considered to be one of the next disruptive usages of computers in the future. To tackle the complexity of reasoning tasks, these systems leverage techniques like models@run.time to structure and process measured information in efficient context representations. However, today's computation theory is based on the discretization of observable data into a sequence of events, augmented by timestamps. Nevertheless, the amount of expected data in the IoT domain makes the construction of reasoning contexts through the enumeration of all timestamped measurements challenging. Beyond models@run.time, this discrete representation for IoT is depicted by Jacobs *et al* [12] as the *pathology of Big Data*.

In this paper we claim that a discrete data representation is not optimal for all attribute types in models, especially for data which is naturally continuous. Since models should be *simpler, safer and cheaper than the reality* [15] we introduce a new meta attribute type for continuous IoT related data. By leveraging our previous results on model element versioning and the state-of-the-art time series compression algorithms, we enable models@run.time to handle millions of volatile data aside traditional discrete data such as string and relationships. We enhanced and adapted signal processing and time series techniques to allow models to go beyond value enumeration capabilities. We integrated our algorithm based on polynomial representation into the Kevoree Modeling Framework to seamlessly and efficiently store continuous values in any models.

During an experimental validation we demonstrated the suitability of this approach to build context models handling millions of values from various IoT datasets. The various benchmarks conducted in this work, lead to the conclusion that this representation can enhance, with a very significant factor (from 20 to 1000), primary operations for intelligent systems such as read or write over a context model.

Acknowledgment

This research is supported by the National Research Fund Luxembourg (grant 6816126) and Creos Luxembourg S.A. and the CoPAInS project (CO11/IS/1239572).

References

[1] S. P. Gartner, "The five smart technologies to watch," 2014.
[2] G. S. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.
[3] N. Bencomo, R. B. France, B. H. C. Cheng, and U. Aßmann, Eds., *Models@run.time - Foundations, Applications, and Roadmaps [Dagstuhl Seminar 11481, November 27 - December 2, 2011]*, ser. Lecture Notes in Computer Science, vol. 8378. Springer, 2014.
[4] H. Demirkan and D. Delen, "Leveraging the capabilities of service-oriented decision support systems: Putting analytics and big data in cloud," *Decision Support Systems*, vol. 55, no. 1, pp. 412–421, 2013.
[5] S. LaValle, E. Lesser, R. Shockley, M. S. Hopkins, and N. Kruschwitz, "Big data, analytics and the path from insights to value," *MIT Sloan Management Review*, vol. 21, 2013.
[6] M. L. Hogarth, "Does general relativity allow an observer to view an eternity in a finite time?" *Foundations of physics letters*, vol. 5, no. 2, pp. 173–181, 1992.
[7] PlanetCassandra, "Getting started with time series data modeling." [Online]. Available: http://planetcassandra.org/getting-started-with-time-series-data-modeling/
[8] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.
[9] S. Gibbs, C. Breiteneder, and D. Tsichritzis, *Data modeling of time-based media*. ACM, 1994, vol. 23, no. 2.
[10] oreilly, "The re-emergence of time-series," 2013. [Online]. Available: http://radar.oreilly.com/2013/04/the-re-emergence-of-time-series-2.html
[11] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé *et al.*, "Ibm streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7–1, 2013.
[12] A. Jacobs, "The pathologies of big data," *Communications of the ACM*, vol. 52, no. 8, pp. 36–44, 2009.
[13] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE, 2001, pp. 289–296.
[14] T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, O. Barais, and Y. Le Traon, "A native versioning concept to support historized models at runtime," in *17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*. Springer, 2014, pp. 252–268.
[15] J. Rothenberg, L. E. Widman, K. A. Loparo, and N. R. Nielsen, "The nature of modeling," in *Artificial Intelligence, Simulation and Modeling*, 1989, pp. 75–92.
[16] InfluxDB, 2014. [Online]. Available: http://influxdb.com
[17] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *Knowledge and information Systems*, vol. 3, no. 3, pp. 263–286, 2001.
[18] K.-P. Chan and A.-C. Fu, "Efficient time series matching by wavelets," in *Data Engineering, 1999. Proceedings., 15th International Conference on*. IEEE, 1999, pp. 126–133.
[19] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth, "Rule discovery from time series." in *KDD*, vol. 98, 1998, pp. 16–22.
[20] M. Kristan and A. Leonardis, "Online discriminative kernel density estimator with gaussian kernels," *Cybernetics, IEEE Transactions on*, vol. 44, no. 3, pp. 355–365, 2014.
[21] P. Abeles, "Efficient java matrix library," 2010.
[22] W. Philips and G. De Jonghe, "Data compression of ecg's by high-degree polynomial approximation," *Biomedical Engineering, IEEE Transactions on*, vol. 39, no. 4, pp. 330–337, 1992.
[23] S. Dusan, J. L. Flanagan, A. Karve, and M. Balaraman, "Speech compression by polynomial approximation," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 15, no. 2, pp. 387–395, 2007.
[24] H. Chen, J. Li, and P. Mohapatra, "Race: Time series compression with rate adaptivity and error bound for sensor networks," in *Mobile Ad-hoc and Sensor Systems, 2004 IEEE International Conference on*. IEEE, 2004, pp. 124–133.
[25] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 29–42.