TimeTravel:Model-based Integration of Past & Future Data

Mohamed E. Khalefa [#1], Ulrike Fischer [*2], Torben Bach Pedersen [#3]

[#]*Department of Computer Science, Aalborg University, 9220 Aalborg, Denmark*
[1]mohamed@cs.aau.dk
[3]tbp@cs.aau.dk

[*]*Database Technology Group, Dresden University of Technology, 01062 Dresden, Germany*
[2]ulrike.fischer@tu-dresden.de

***Abstract*— Seamless integration for past and (forecasted) future values of time series is crucial for advanced application domains. For example, in energy domain, it is important to balance the production and consumption electricity. Typically, for these applications, it is more useful to provide immediate approximate answers for queries than later exact answers. For future data points, we can only compute approximate answers. The main idea behind in our proposed system, TimeTravel, is to compactly represent time series as models. By using models, we can answer queries approximately on past and future data with error guarantees (absolute error and confidence) one order of magnitude faster than when accessing the time series directly. In addition, it efficiently supports exact historical queries by only accessing relevant portions of the time series. This is unlike existing approaches, which access the entire time series to exactly answer the query.**

To realize this system, we propose a novel hierarchical model index structure. As real-world time series usually exhibits seasonal behavior, models in this index incorporate seasonality. To construct a hierarchical model index, the user specifies seasonality period, error guarantees levels, and a statistical forecast method. As time proceeds, the system incrementally updates the index and utilizes it to answer approximate and exact queries. TimeTravel is implemented into PostgreSQL, thus achieving complete user transparency at the query level.

Extensive experimental analysis show to illustrate the speed up for approximate and exact queries with varying the error guarantees.

I. Introduction

Time series can be encountered in many applications, including financial (e.g., stock price [14]) and scientific database (e.g., sensor data for weather information [11] or environmental data). Time series usually exhibit one or more seasonal behaviors. For example, power consumption rises in winter (e.g., heating) and during the day and falls in the summer and at night. Modeling has been extensively used in different applications including forecasting, estimating, approximating, neuroscience, and rendering graphics for computer games. Several forecasting methods have been proposed for different applications such as neural network average for stock trading [14], effect-graph for foreign politics [7] and ARIMA [4] in energy domain [11]. Forecasting is essential for decision making applications (e.g., planning of production batches). To model time series, many techniques have been proposed in the literature including Piecewise Aggregate Approximation (PAA), Adaptive Piecewise Constant Approximation (APCA), Chebyshev polynomials (CHEB), Symbolic Aggregate approXimation (SAX), Indexable Piecewise Linear Approximation (IPLA). These approaches mainly address mining and similarity queries. A comparative study for their performance for various time series in presented in [9]. IPLA slightly outperforms other methods in lower bounding for DTW distance function which is mainly used for similarity queries.

Traditionally, a time series can be decomposed into a trend component, a seasonal component, and a local (i.e., stationary) component [12]. Statistical forecasting methods (e.g., ARIMA [4]) use these components to accurately compute forecasted values. For our purpose, we abstract a forecast method as a function which takes forecast parameters and states (i.e., past values) and outputs future values. We use this decomposition to represent the time series over the past and the future. Consider a motivating example from the energy domain: the power grid operator wants to investigate when the power consumption exceeded (or will exceed) a certain threshold (e.g., 90%) of the network over the previous and upcoming weeks. Several approaches (e.g.,[1] and [13]) use error and confidence to speedup the query execution. Consider for our example, an approximate answer with 1% error and 95% confidence is adequate. A naive approach would scan the data points over the previous week, optimize parameters for a forecast method, and predict the time series for the next week. Finally, the query discards values that are lower than the specified threshold. Our proposed system performs this more intelligently, as will be explain next.

Throughout the paper, we use our motivating example shown in Figure 1. Figure 1a gives 50 values which is based on real world power consumption in UK grid. We can decompose this time series into three components (two seasonal and one trend components) as shown in Figure 1b. The seasonal components have periods of 10 and 4 cycles. By storing the decomposed complements instead of the original timeseries, we reduce the storage requirements. In the toy example, instead of storing the 50 values of the original time series, we need only store 19 values to exactly represent the time series (i.e., with 0% error and 100% confidence)(5 values for trend and 10 and 4 seasonality periods) which corresponds a compression ratio of 42%. Moreover, we can forecast the time

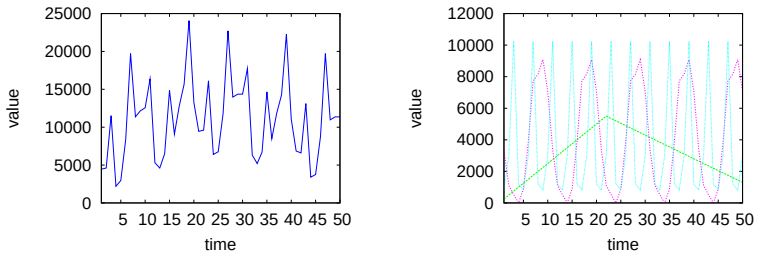

Fig. 1. Motivating Example

series by exaggerating the decomposed components.

To this end, we introduce a TimeTravel, an efficient DBMS system for seamless integration of past and future time series values, supporting two types of queries: (a) approximate queries on past and future within user-specified error bounds (i.e., absolute error and confidence), and (b) exact historic queries. We can support point, range, aggregate and join queries. Unlike, previous approaches, we utilize trend and seasonality components to build models over the underlying time series to support . Ignoring seasonality components results in low compression ratio as we will discuss later. Here, past and future data is treated similarly, allowing seamless integrated querying. The main difference of future data compared to past is that the (estimated) error is typically higher. To organize these models and efficiently support queries with various error guarantees, we introduce a novel index structure, denoted as *hierarchical model index*. The upper levels in the index are more "coarse-grained", representing the underlying time series with a higher error and a lower confidence and fewer model segments compared to the lower levels which are more accurate. Please note that confidence while building models to reduce the effect of *outliers* on the quality of models.

For approximate historical queries (e.g., last week in our motivating example), we use the highest-level models to calculate an approximate answer. If the answer violates the user requirements on error and confidence, we consult relevant models at lower levels. We continue traversing the hierarchical model index until either the error guarantee given by user is met or the underlying time series is accessed. In addition, TimeTravel efficiently answers exact queries by utilizing the index to find the relevant portions in the time series. For example, considering MIN aggregation queries, we only access the portion of the time series where the minimum value might exist. For future queries (e.g., next week in our motivating example), we use the model index to retrieve forecast states which is supplemented to statistical forecast methods. We meet the user required error guarantees on future data by (a) controlling the error and confidence of the retrieved forecast states and (b) optimizing the forecast method parameters.

To build the hierarchical model index, the user specifies hints for: (1) seasonality periods, (2) error guarantees levels, and (3) the statistical forecast method (e.g., ARIMA). We recursively divide the time series into non-overlapping intervals and build a model on each interval until all error requirements are satisfied. Over time, *new* values are added to the time series, and we incrementally update the parameters for the forecast method and add models to the hierarchical model index.

In the rest of this paper, first we formalize the problem statements in Section III. Section II provides an overview of the our proposed system. Section **??** describes building the hierarchical model index, and compression Section **??** gives the details of query optimization and processing for approximate and exact queries. The related work is presented in Section VI. Section **??** gives our experimental analysis for our proposed system using both real-world time series data and synthetic data.

II. TimeTravel System Overview

Our prototype system, TimeTravel, is implemented *inside* the PostgreSQL database engine. For each time series, our proposed system builds a separate model index. Figure 2 illustrates the main modules and data structures of TimeTravel. Time series is stored as an array indexed by time. A time series can have an arbitrary number of seasonality periods (e.g., the shown time series exhibits two seasonality periods P_1 and P_2). The hierarchical model index is presented in the top center of Figure 2. Each model segment, MS for short, at level l has zero or more children segments at level $l-1$. Each child segment improves its parent's representation of the underlying time series by building either (a) a more detailed model over the time series, or (b) models of the error of the parent model. We denote the latter models as ΔModels. For query optimization, we store statistics about the index in the system catalog. Specifically, we store the number of model segments, their average size, and the maximum error and minimum confidence for each level. These statistics are needed to estimate the expected cost for the query. The main modules can be presented as follow:

Building Module. The purpose of this module is to take user hints and time series as inputs, and output a compatible hierarchical model index. We use heuristics to determine the model parameters. This module is invoked offline and presented in Section IV.

Compression Module. This module reduces the required storage space for hierarchical model index by finding similar model segments, based on trend and seasonality components, and combining them into fewer segments.

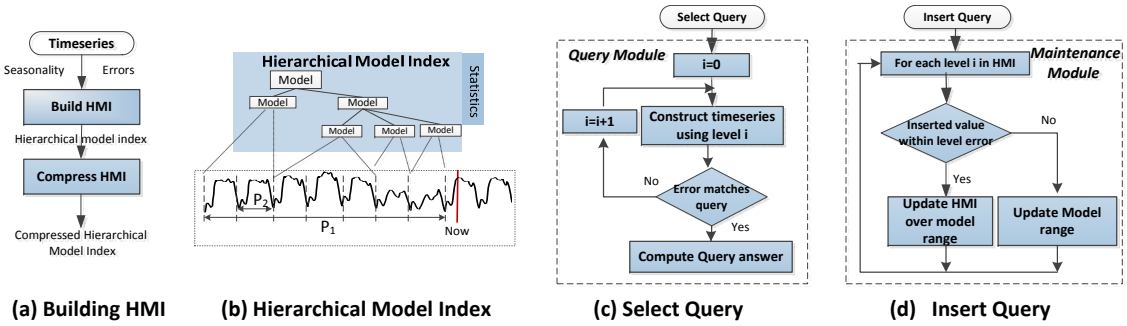

Fig. 2. TimeTravel System Overview

Query Processing Module. We extend the query processor and optimizer of PostgreSQL to support approximate queries over future and past data and exact queries over the past data. To answer historical queries, it traverses down the model index. The Forecasting module is used to retrieve the predicted the time series. The query processor supports point, range, aggregate, and join queries, as discussed in Section V.

Forecasting Module. The forecasting module is responsible for: (1) predicting the values of time series for a given future interval, (2) estimating the error and the confidence of the forecasted values, and (3) reestimating forecast method parameters. It uses the query processing module to retrieve the forecast method states. The error of the forecasted data [4] depends on (1) the forecast model and the length of forecast data and, (2) the confidence interval of the retrieved states. Further details are presented in [10].

Maintenance Module. The Maintenance module maintains hierarchical model index when new values are added to the time series. For each level in the index, we update the last segment (i.e., the rightmost segment in the figure) with the added values. If the error and the confidence of the updated segment violates the model error guarantee, we construct a new segment over the interval of the last model and added values, using the building module, and substitute it for the old one.

III. Problem Formulation

In this section, we give definitions for our proposed problem. Throughout the paper, we assume a mathematical model to represent the underlying timeseries. More specifically, to compactly represent a timeseries TS using model M_{TS}, we either: (1) additively decompose TS into $trend$, $seasonal$ and $error$ using STL algorithm [6]. Then, we compute a separate model for each component, or (2) use explicit values, average value, or regression line.

First, we can formally define a timeseries as follow:

Definition 1: TimeSeries: A time series TS is an ordered temporal sequence. TS= (ts_1, ts_2, ..., ts_n), where n is time series length, and $\forall_i\ ts_i \in \mathbb{R}$. Timeseries have a set of periods $\mathcal{P}$= {1, P_1, ... P_m} where $m > 0$, and $\forall_i\ P_i$ is a sequence on the form k_l* P_l, s.t., $k_l \in \mathbb{N}$ and $l < i$.

Timeseries may have one or more seasonal periods. For example, a time series may have yearly, weekly and daily periods. Weekly periods may divided into five work-days and two weekend-days. Alternatively, we can divide yearly period directly into work days and week-end days.

The mathematical model can be defined as follow:

Definition 2: TimeSeries Model: Model M_{TS} for time-series TS with t, s, and e as trend, seasonal and error additive components for timeseries TS. i.e.,

$$M_{TS}(t) = \begin{cases} TS(t) & \text{;Explicit values} \\ a * t + b & \text{;Regression line} \\ M_t(t) + M_s(t) + M_e(t) & \text{;Decomposed} \end{cases}$$

The model M does not perfectly match the underlying data, the deviation between the values computed using the model and actual values is referred to as model errors. The confidence of a model M is the probability that model errors is below certain error level. For example, for 100% confidence, the error level must match the maximum model errors. We can formally define a model matching for error guarantee, as follow:

Definition 3: The model M matches error guarantee (ϵ, δ) iff: Pr $\{\ |(M_{TS}(t) - TS(t)| \leq \epsilon\ \} \geq \delta$ where $TS(t)$, $M_{TS}(t)$ is the actual and model time series value at t, respectively.

The goal of our proposed solution is to find models for a timeseries which meet all error guarantees while reducing the size. It can be formally presented as follow:

Definition 4: Problem statement: Given: a timeseries TS, and error guarantees: (ϵ_1,δ_1),(ϵ_2,δ_2)..., (ϵ_m,δ_m). Find models M_1, M_2, ..., M_k on the entire or interval of time series TS. $\forall$ timestamp t in timeseries, $\exists$ j and l s.t. M_j matches error guarantee (ϵ_l,δ_l) while minimizing the size of all models, i.e., $Min(\sum_{i=1}^{k} size(M_i))$.

IV. The Hierarchical Model Index

In this section, we present building and compressing the Hierarchical Model Index and storing it into PostgreSQL. Finally, we describe incrementally updating the index with the inserting of new values to the time series.

A. Building the Hiericahial Model Index

We now describe constructing a hierarchical model index over the time series. The pseudo code is given in Algorithm 1. The algorithm takes two inputs: (1) timeseries TS with seasonal periods hierarchy (we use user hints for seasonality period) and (2) a set of error guarantees, $error_g$, represented as

pair of error and confidence. The algorithm finds a hierirehcal model index for the given timeseries such that for each error guarantee, there exists a model (or more) in the index which meets this requirements for the entire timeseries.

Initially, we build one model over the entire time series. This model represents the root of the index structure. All other models of timeseries (with tighter error gurantess) are either direct or non-direct children of the root node. To find the best model representation over entire (or segment of) time series, we use a utility function defined as the size of the model multiplied by the δ-error (i.e., the probability that the absolute error of the model is bounded by δ). The utility of the time series is defined to be infinity. The best model for a time series is defined as the model with the smallest utility. The algorithm first starts with getting the least error guarantee (ϵ, δ). (Line 2 in Algorithm 1) Then, we iterate over all periods in the seasonal hierarchy to find best model representation. (Lines 4-18 in Algorithm 1) We decompose the timeseries TS into trend t, seasonal s, and error e using STL algorithm [6]. Then, using FindPossibleTrend and FindPossibleSeasonality (the pseudo code is shown in Algorithm 2 and Algorithm 3), we find possible representations for the trend and seasonality components t , s respectively. For each possible representations for trend from set $\hat{T}$ and seasonality from set $\hat{S}$ (Line 9 in Algorithm 1), First, we compute the error $\hat{e}$ by subtracting the original timeseries from the trend and seasonality representations. (Line 9 in Algorithm 1) Then, we use DivideError algorithm to divide the model error, i.e., $\hat{e}$, into non-overlapping intervals using the error ϵ. The intervals are divided into three categories: (1) the error of the interval is less than ϵ, (2) the difference between the minimum and maximum error is bounded by the error ϵ, and (3) the error of the interval exceeds the threshold error ϵ. (Line 11 in Algorithm 1) The size of model is computed as the sum of the size of trend and seasonality representations $|\hat{t}|$ and $|\hat{s}|$, and size needed to store the intervals and the minimum and maximum error bound for the model. (Line 12 in Algorithm 1) Then, we compute the utility of the model. We use the utility to compare the model with the previous models, keeping the model with largest utility. (Lines 14- 16 in Algorithm 1)

After finding the model with the best utility, we iterate over all intervals which is computed earlier using DivideError algorithm in Line 11 in Algorithm 1 (Line 19 in Algorithm 1). For each interval, first we remove from the error guarantees set, i.e., $error_g$, all the error guarantees that are matched. Then, if the interval does not meet all the error guranete, we recursively build two models over the timeseries and errors in interval I denoted as $model_I$ and $\Delta model_I$ respectively. (Lines 22- 23 in Algorithm 1) We add the model of interval I with best utility as a child to the model. (Lines 24- 25 in Algorithm 1) Finally, we return the computed model.

Processing Trend Component We examine two trend representations based on means and linear regressions. The pseudo code is presented in Algorithm 2. The algorithm takes two inputs: (1) the trend component and (2) the length of the seasonal period. Initially, if the seasonality period is greater

Algorithm 1 Creating a Hierarchical Model Index

Input TS timesreies, $Error_g$
1: **procedure** BUILDMODEL(TS, $Error_g$)
2: $(\epsilon, \delta) \leftarrow Last(Error_g)$
3: $model \leftarrow NULL$ ▷ best model yet found
4: **for each** $period \in TS.period$ **do**
5: $(t, s, e) \leftarrow Decompose(TS, period)$
6: $\hat{T} \leftarrow FindPossibleTrend(t, period)$
7: $\hat{S} \leftarrow FindPossibleSeasonality(s, period)$
8: **for each** $(\hat{t}, \hat{s}) \in \hat{T} \times \hat{S}$ **do**
9: $\hat{e} \leftarrow TS - (\hat{t} + \hat{s})$
10: $Error_\delta \leftarrow (\delta * 100)^{th} ofabs(\hat{e})$
11: $\mathcal{I} \leftarrow DivideError(\hat{e})$
12: $size \leftarrow |\mathcal{I}| + |\hat{t}| + |\hat{s}| + 2$
13: $utility \leftarrow size * Error_\delta$
14: **if** $model.utility < utility$ **then**
15: $model \leftarrow (\hat{t}, \hat{s}, \hat{e}, \mathcal{I})$
16: **end if**
17: **end for**
18: **end for**
19: **for each** $I \in \mathcal{I}$ **do**
20: $\mathcal{E}_g \leftarrow Error_g - Matched(model_{best}, I)$
21: **if** $\mathcal{E}_g ! = \phi$ **then**
22: $model_I \leftarrow BuildModel(TS[I], \mathcal{E}_g)$
23: $\Delta model_I \leftarrow BuildModel(\hat{e}[I], \mathcal{E}_g)$
24: $m \leftarrow Best_{utility}(\Delta model, model_I)$
25: $model.addChild(m)$
26: **end if**
27: **end for**
28: **return** model
29: **end procedure**

Algorithm 2 Find Possible Representations for Trend

1: **procedure** FINDPOSSIBLETREND(T, $period$)
2: $\hat{T} \leftarrow \{\}$
3: **if** $period > 0$ **then**
4: // Use Means
5: $\hat{T} \leftarrow \hat{T} \bigcup \{\frac{\Sigma_{i=1}^{period} T_i}{period}, \frac{\Sigma_{i=period+1}^{2*period} T_i}{period}, \dots\}$
6: $\hat{T} \leftarrow \hat{T} \bigcup \{Mean(T[1 : period]), Mean(T[period + 1 : 2 * period]), \dots\}$
7: **end if**
8: // Use regression
9: $\hat{T} \leftarrow \hat{T} \bigcup ChebyshevReg(TS)$
10: **return** $\hat{T}$
11: **end procedure**

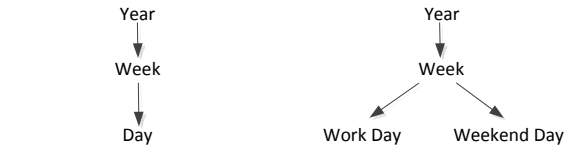

Fig. 3. Possible Hints for Seasonal Hierarchy

than zero, we present the trend as a vector of values over each seasonal period. (Lines 3-7 in Algorithm 2) We use the linear Chebyshev regression method [15] to approximate the trend component T. (Lines 9 in Algorithm 2) The Chebyshev regression method is efficiently computable and minimizes the largest deviation (i.e. error) from the original curve (i.e., time series) among the polynomials of the same order. Finally, the algorithm returns possible representations for the trend components. (Line 10 in Algorithm 2)

Processing Seasonality Component Storing the value of each data point in the seasonality component may be expensive (e.g., yearly seasonality). We may approximate the seasonality component using the user hints by recursively calculating the trend and seasonality using smaller seasonality periods. The pesduo code for finding possible representations in given in Algorithm 3. The algorithms takes two inputs: (1) the seasonality component, and (2) the period seasonality. Please note for this algorithm we need the seasonality period hierarchy. Initially, a possible representation is to directly store data points of the seasonality component, S. (Line 2 in Algorithm 3) We explore this hierarchical seasonal information to find different representations for the seasonal component. Therefore, we iterate over all the seasonality periods that are smaller than the seasonality period of S. (Line 4 in Algorithm 3) If the seasonality period consists of more than one seasonality period, for example a weekly seasonality in figure 3b consists of workday and weekend day, we divide the seasonality into partitions accordingly. (Lines 4-8 in Algorithm 3) For each seasonality period S, we decompose it into three components $(S.t, S.s, S.e)$. Then, we use FindPossibleTrend over the computed trend $S.t$ and recursively use FindPossibleSeasonality for the computed seasonality component $S.S$. For each possible representation for trend and components, it is added to the possible seasonal representations (Lines 9-12 in Algorithm 3) Finally, we return the set of possible seasonality representations.

Algorithm 3 Find Possible Representations for Seasonality

1: **procedure** FINDPOSSIBLESEASONALITY(S, $period$)
2: $\hat{S} \leftarrow \{S\}$
3: **for each** $p \in TS.periods, s.t., p < period$ **do**
4: **if** p contains more than one period **then**
5: $\mathcal{S} \leftarrow DivideTimeseries(S, p)$
6: **else**
7: $\mathcal{S} \leftarrow S$
8: **end if**
9: **for each** $S \in \mathcal{S}$ **do**
10: $(S.t, S.s, S.e) \leftarrow Decompose(TS, p.period)$
11: $\hat{S} \leftarrow \hat{S} \cup FindPossibleTrend(S.t) \times FindPossibleSeasonality(S.s)$
12: **end for**
13: **end for**
14: **return** $\hat{S}$
15: **end procedure**

Algorithm 4 Compressing The Hierarchical Model Index

1: **procedure** COMPRESSINGMODELINDEX(HIM, $error_g$)
2: **for each** $similar\ models\ M_i, M_j \in \mathcal{M}$ **do**
3: $TS_j^i \leftarrow M_i(TS_j)$
4: **if** $\epsilon(TS_j^i)$ is within error guarantee of M_j **then**
5: Replace M_j with M_i in HIM
6: Delete M_j
7: **else if** $\epsilon(TS_i^j)$ is within error guarantee of M_i **then**
8: Replace M_i with M_j in HIM
9: Delete M_i
10: **else**
11: $M_{ij} \leftarrow BuildModel(TS_i \cup TS_j, error_g)$
12: **if** $error of M_{ij} matches M_i and M_j$ **then**
13: Replace M_j and M_i with M_{ij} in HIM
14: Delete M_i; Delete M_j
15: **end if**
16: **end if**
17: **end for**
18: **end procedure**

B. Index Compression

We can reduce the the size of hierarchical model index while meeting the error guarantees. This is based on that the error and confidence for some models are lower than the user-specified error guarantees. Several algorithms

Algorithm 4 gives the pseudo code for compressing the hierarchical model index, it takes the model index and error guarantees as inputs. First, we iterate through similar models in the model index. The similarity for two models M_i and M_j are in relative to trend and seasonality representations. Please note that we normalize the representations for both models to find the similarity. (Lines 2-3 in Algorithm 4) Then, we use M_i to compute data points over the range of model M_j. If the error and confidence for the computed data points is within the error guarantee for model M_j, we drop M_j from the HIM, and use denormalized M_i instead. (Lines 4-6 in Algorithm 4) Similarly, we check if model M_j can be used to compute the data points of model M_i. (Lines 7-9 in Algorithm 4) Otherwise, we build new model M_{ij} over data points of models M_i and M_j. M_{ij} can be used only if the error guarantees are matched for both M_j and M_i. (Lines 11-14 in Algorithm 4)

C. Storage

A system catalog table, PG_{HIM}, is added to store the information needed about hierarchical model indexes. We extend the relation to store the model index, the first tuple in the first page corresponds to the root of the model index. Each entry in the catalog table contains: (1) The attribute in the base relations, and (2) the relation used to represent the hierarchical model index. In addition, we store in system catalog, PG_{eg}, for each error guarantee, the needed number of page access in its model index.

To natively store the described model index in PostgreSQL, we extended the page layout to be able to efficiently store this recursively data structure. We store the model header which consists of: (1) model type (i.e., delta model, normal model), (2) the minimum and maximum error, (3) the range where the model is valid, (4) the number of children models with pointer to model index, (5) the trend representation, and (6) the seasonality representation which may be recursively represented.

V. Query Processing

In this section, we describe query processing and optimization for point, range, aggregate, and join queries over past or future data for approximate and exact queries. The underlying time series is stored as an array indexed by the time attribute.

To completely incorporate the model index into Postgres, we extend the query optimizer to estimate the expected cost of queries using the proposed model index. Specifically, we add to the system catalog statistical information for average model size, number of model segments, the span of the seasonality component, and the maximum error and minimum confidence for each level in the model index. Moreover, we store the number of data points of the time series that fits in one page. By using this information, the query optimizer estimates the cost of executing a query using the model index or directly using the stored time series, and creates the query plan accordingly. As mentioned in [16], the dominant cost of forecast queries is re-estimating the forecast model parameters. Unfortunately, the burden of this cost cannot be avoided. However, we only reestimate the parameters if necessary[10], and the cost is amortized over the query workload. For simplicity, we do not include it in our cost model.

The general idea of query processing is to construct an approximation for the time series using the models and the error guarantee. Each value is represented as an uncertain range $[v+l, v+u]$, where v is the value from the model, and l and u are the lower and upper bound of the error, respectively. The approximation can be arbitrarily improved by traversing down the index. An important optimization is that we use the model parameters, span of seasonality, and error to find the relevant portion of the time series.

Point Queries. As the underlying time series is stored as an array, we need at least one page access to find the value of the time series for any point in the past, with 0% error and 100% confidence, while for forecast queries, we access the forecast model parameters and states (i.e., historic values). We estimate the cost of finding the values of the historic data either using the time series or using the index structure.

Range Queries. We use the system catalog statistics to estimate the cost of a range query. We accomplish this by computing an upper bound on the number of model segments in the requested range at each level in the model index. While the cost of directly using the time series is $\frac{r}{b}$, where b is the number of data points per page, and r is the query range. To compute any future data point for forecast queries, we need to retrieve the relevant forecast states, i.e., to answer a range query Q in the future, we need to compute range queries over the past to get all states. To this end, we can estimate the expected cost for forecast range queries by either using the model index or accessing the underlying time series directly. Processing a range query Q can be presented as follows: we traverse down the model index, discarding model segments that do not overlap with the specified range r until we reach level l that matches the user requirements of error and confidence.

Aggregate Queries The query processing and cost estimation for an aggregate query depends on the type of aggregate function used. For MIN/MAX aggregates, we use the approximated representation of the time series to identify possible intervals (i.e., ranges) within the query range where the minimum (maximum) may exist, discarding most of the range of the query. For SUM/AVG aggregates, a nice property of linear regression is that the total sum of the errors equals zero. Hence, while traversing the hierarchical model index, we discard an interval l from further processing, if we encounter a model segment m over interval l that fully overlaps with the range of the query.

Join Queries. For simplicity, we limit our discussion to equi-join queries. Joining the time series x and y over the time attribute can be performed as follows: we progressively traverse the indexes for time series x and y. If the error of the combined results exceeds the query requirements, we increase the accuracy of join results by traversing down the index with the maximum error. For joining time series on a value attribute, we build an approximate representation for each time series using the model index. For each potential intersection (i.e., join output), we eliminate false positives by traversing down the corresponding model index.

VI. Related Work

MauveDB [8] proposed querying models using a relational framework for sensor data. FunctionDB [19] supports regression functions to represent the data. Queries on these function are processed using an algebraic solver. Pluse [2] is a continuous stream query 'processor that can fit one-dimensional functions. It finds the query results by computing the algebraic solution to systems of equations. It back-propagates the query result to validate the error on the input. In comparison, in TimeTravel, we directly associate the model with error and hence can bound the error of the answers. To answer approximate queries Shatkay [17] proposed an algorithm to fit a curve using a set of lines or Bezier curve while bounding the

error. All of these approaches uses only regression functions. Ignoring seasonality significantly increases the number of needed regression functions. MauevDB and functionDB do not give any error guarantee, nor support forecast queries. In comparison, TimeTravel supports more powerful functions, seasonality, and error guarantees.

On the other hand, ISAX [18], [5] and data clipping [3] approximate the time series by mapping each value in the time series to a value in a much smaller domains (e.g., a domain of two values in [3]). Unlike our approach, the error of depends on the cardinality of the mapped domain and the compression ratio.

similarity not

VII. Experiments

Building model tree and compressing scaliblity done inceremental (not yet) Query done paritally Past future combined add an experiemnts with the number of values size for each level

VIII. Conclusion

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. *SIGMOD Rec.*, 28(2):275–286, June 1999.

[2] Y. Ahmad, O. Papaemmanouil, U. Çetintemel, and J. Rogers. Simultaneous equation systems for query processing on continuous-time data streams. In *ICDE*, pages 666–675, 2008.

[3] A. J. Bagnall, C. A. Ratanamahatana, E. J. Keogh, S. Lonardi, and G. J. Janacek. A bit level representation for time series data mining with shape based similarity. *DMKD*, 13(1):11–40, 2006.

[4] G. E. P. Box and G. M. Jenkins. *Time Series Analysis, Forecasting, and Control*. Holden-Day, 1976.

[5] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. *i*sax 2.0.

[6] R. Cleveland, W. Cleveland, J. McRae, and I. Terpenning. Stl: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics*, 6(1):3–73, 1990.

[7] B. B. de Mesquita. *Forecasting Policy Futures*. Ohio State Univ, 2004.

[8] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84, 2006.

[9] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, Aug. 2008.

[10] U. Fischer, F. Rosenthal, and W. Lehner. F2DB: the flash-forward database system. In *ICDE*, pages 1245–1248, 2012.

[11] R. Kavasseri and K. Seetharaman. Day-ahead wind speed forecasting using f-arima models. *Renewable Energy*, 34(5):1388–1393, 2009.

[12] M. Kendall and A. Stuart. *The Advanced Theory of Statistics*, volume 3. Griffin, 1983.

[13] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Skyline query processing for uncertain data. In *CIKM*, pages 1293–1296, 2010.

[14] K. Kohara, T. Ishikawa, Y. Fukuhara, and Y. Nakamura. Stock price prediction using prior knowledge and neural networks. *Intelligent Systems in Accounting, Finance & Management*, 6(1):11–22, 1997.

[15] J. C. Mason and D. Handscomb. *Chebyshev Polynomials*. Chapman & Hall, 2003.

[16] F. Rosenthal and W. Lehner. Effcient in-database maintenance of arima models. In *SSDBM*, pages 537–545, 2011.

[17] H. Shatkay and S. B. Zdonik. Approximate queries and representations for large data sequences. In *ICDE*, pages 536–545, 1996.

[18] J. Shieh and E. J. Keogh. *i*SAX: indexing and mining terabyte sized time series. In *KDD*, pages 623–631, 2008.

[19] A. Thiagarajan and S. Madden. Querying continuous functions in a database system. In *SIGMOD*, pages 791–804, 2008.