Model-based Integration of Past & Future in TimeTravel

Mohamed E. Khalefa[1] *, Ulrike Fischer[2], Torben Bach Pedersen[3], Wolfgang Lehner[4]
[1,3] Department of Computer Science, Aalborg University, 9220 Aalborg, Denmark
[2,4] Database Technology Group, Dresden University of Technology, 01062 Dresden, Germany
[1,3]{mohamed, tbp}@cs.aau.dk, [2,4]{ulrike.fischer, wolfgang.lehner}@tu-dresden.de

ABSTRACT

We demonstrate TimeTravel, an efficient DBMS system for seamless integrated querying of past and (forecasted) future values of time series, allowing the user to view past and future values as one joint time series. This functionality is important for advanced application domain like energy. The main idea is to compactly represent time series as models. By using models, the TimeTravel system answers 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. In the demo, we show the easy building of a hierarchical model index for a real-world time series and the effect of varying the error guarantees on the speed up of approximate and exact queries.

1. INTRODUCTION

Time series can be encountered in many applications, including financial (e.g., stock price [9]) and scientific database (e.g., sensor data for weather information [7] 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 afternoon and falls in the summer and at night. Traditionally, a time series can be decomposed into a trend component, a seasonal component, and a local (i.e., stationary) component [8]. Statistical forecasting methods (e.g., ARIMA [3]) 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. 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 months. An approximate answer with 1% error and 95% confidence is adequate. A naive approach would scan the data points over the previous month, optimize parameters for a forecast method, and predict the time series for the next month. Finally, the query discards values that are lower than the specified threshold. Our system performs this more intelligently, as will be explain next.

We demonstrate TimeTravel, an efficient DBMS system for seamless integrated querying 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 use trend and seasonality to build models over the underlying time series. 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, 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. For approximate historical queries (e.g., last month 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 month 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

*Work partly done while visiting Dresden University of Technology

Proceedings of the VLDB Endowment, Vol. 5, No. 12
Copyright 2012 VLDB Endowment 2150-8097/12/08... $ 10.00.

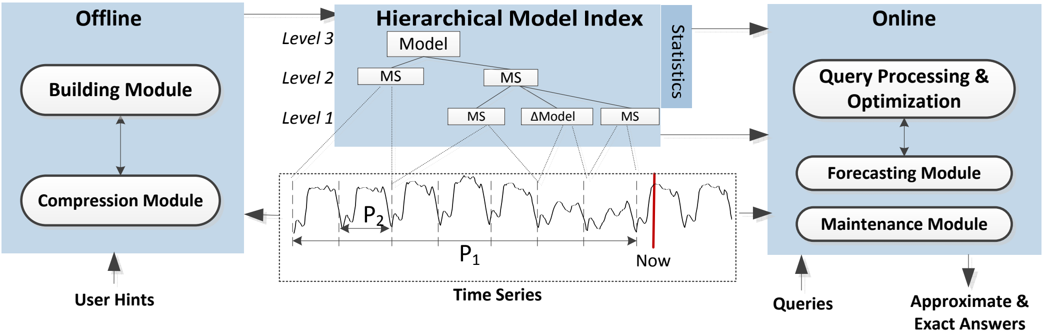

Figure 1: TimeTravel Overview

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 the hierarchical model index.

2. TIMETRAVEL OVERVIEW

Our prototype system, TimeTravel, is implemented *inside* the PostgreSQL database engine. The user specifies a number of time series and the system builds a separate model index for each. Figure 1 illustrates the main modules and data structures of TimeTravel. Time series is stored as an array. 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 1. 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 3.

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.

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 4.

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 [3] 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 [6].

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.

3. THE HIERARCHICAL MODEL INDEX

We now describe how to construct a hierarchical model index over the time series. Initially, we build one model segment over the entire time series. Based on the model error, we divide the time series into non-overlapping sub-intervals I_i (e.g., using approaches in [14, 12]). The user specifies hints for the seasonal period hierarchy. For an hourly time series, Figure 2 shows two possible hints for seasonality. The first hint should be used if the shape of the time series is similar per year, week and day. If the daily shape for weekdays differs from that of weekends, using the second hint reduces the approximation error. To build model segment m over interval l, we first decompose the portion of time series over this interval into *trend* t, *seasonal* s, and *remainder* r components using the longest seasonal period p in the hierarchy, such that the length of the interval is at least twice p. Then, we process each component as described later. To find the best model representation, we maximize a heuristic, H, based on the computed sub-intervals and the size of the model. Heuristic, H, is defined as $(\sum e_i * |I_i|)/s$, where $|I_i|$ is the length of sub interval I_i, e_i is the number of error guarantees levels which complies with interval I_i, and s is the model size.

Processing Trend Component We use linear Chebyshev regression method [10] to approximate the trend com-

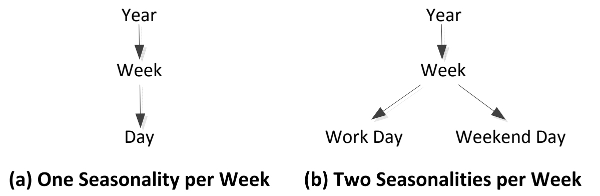

Figure 2: Possible Hints for Seasonal Hierarchy

ponent over the model interval. 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.

Processing Seasonality Component We use the hints for seasonality given by the user to efficiently store the 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. To find the best approximation for seasonality, we utilize the heuristic described above.

Processing Error Component As regression and seasonality model does not perfectly fit the input time series, errors exists. Error is computed as x-($\hat{t}$+$\hat{s}$), where x is time series over interval I, and $\hat{t}$ and $\hat{s}$ are the stored model values for the trend and seasonality components, respectively. We split the error values into at most k intervals on the time dimensions, where k a system parameter representing the maximum fan-out in the model index. For each interval, we can either (1) apply a data clipping techniques (e.g., [4]), (2) store upper and lower error bounds, or (3) fitting a model (i.e., ΔModel) to the error. We choose the representation that maximizes the heuristic describedabove.

4. 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 PostgreSQL, 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 [11], 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[6], 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.

5. RELATED WORK

MauveDB [5] proposed querying models using a relational framework for sensor data. FunctionDB [14] supports regression functions to represent the data. Queries on these function are processed using an algebraic solver. Pluse [1] 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

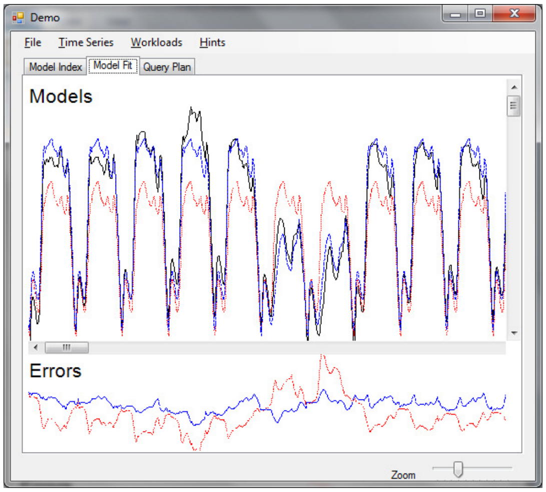

Figure 3: Model Fit & Errors for Levels 1 & 2

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 [12] 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 [13, 4] and data clipping [2] 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 [2]). Unlike our approach, the error of depends on the cardinality of the mapped domain and the compression ratio.

6. DEMONSTRATION SCENARIO

TimeTravel is based on PostgreSQL and integrates the different modules described in this paper. Our demonstration shows building the hierarchical model index for real-world time series, and the effect of varying the error guarantees on the speed up of approximate and exact queries. The user specifies the seasonality hierarchy, error guarantees and the statistical forecast method using a graphical user interface.

Time series We will use several real-world datasets to demonstrate our system. We mainly use the UK household power consumption time series and Australian tourist visitors nights.

Scenarios The prototype is divided in three different aspects: (1) model index view, (2) model fit view, and (3) query plan view. Using the model index view, we illustrate building a hierarchical model index using the user hints. A demo user can browse the resulting hierarchical model index. She may change the heuristic function used, or directly update model parameters (e.g., intervals, regression line, or seasonality). The user can interact with the compression module by combining and separating model segments. For the purpose of the demo, the demo user can submit different query workloads. For any change in the hierarchical model index, we estimate the response time for these query workload, as well as, the total storage space.

By using the model fit view, the demo user can investigate how the model fits the time series using different levels of the hierarchy by displaying the models versus the time series and/or plotting the error component. Figure 3 shows a screenshot of model fit and error using models at level 1 and 2. The time series used is two weeks from UK household power consumption. The black line represents the original time series, while the red dashed lines uses model segments at level 2 which uses the seasonal hint (Figure 2(a)). The blue dotted line shows the models at level 1, which splits the week into workdays and weekend days (Figure 2(b)). As time passes, we show how the expected future data generated from forecast method fits the actual data, and how updates affect the model index.

Last, we will show the query plan for a set of approximate and exact queries given by the user. For each query, we show the tradeoff between query cost and the error guarantee.

7. REFERENCES

[1] 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.

[2] 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.

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

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

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

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

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

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

[9] 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.

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

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

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

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

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