Ulrike Fischer, Frank Rosenthal, Matthias Böhm, Wolfgang Lehner

Indexing forecast models for matching and maintenance

Erstveröffentlichung in / First published in:

IDEAS '10: Fourteenth International Database Engineering & Applications, Montreal 16.-18.08.2010. ACM Digital Library, S. 26-31. ISBN 978-1-60558-900-8.

DOI: https://doi.org/10.1145/1866480.1866485

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 26–31, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866485

Indexing Forecast Models for Matching and Maintenance

Ulrike Fischer, Frank Rosenthal, Matthias Boehm, Wolfgang Lehner
Dresden University of Technology
Database Technology Group
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Forecasts are important to decision-making and risk assessment in many domains. There has been recent interest in integrating forecast queries inside a DBMS. Answering a forecast query requires the creation of forecast models. Creating a forecast model is an expensive process and may require several scans over the base data as well as expensive operations to estimate model parameters. However, if forecast queries are issued repeatedly, answer times can be reduced significantly if forecast models are reused. Due to the possibly high number of forecast queries, existing models need to be found quickly. Therefore, we propose a model index that efficiently stores forecast models and allows for the efficient reuse of existing ones. Our experiments illustrate that the model index shows a negligible overhead for update transactions, but it yields significant improvements during query execution.

Categories and Subject Descriptors

H.2.2 [**Physical Design**]: Access methods; H.2.4 [**Systems**]: Query processing; G.3 [**Probability and Statistics**]: Time series analysis

General Terms

Design, Performance

1. INTRODUCTION

In many domains, gathered data constitutes time series, e.g. sales per month. Such data may be used in decision-making processes that can be improved by a reasonably reliable forecast of the time series, e.g. the planning of production batches is based on anticipated demand.

There has been a recent interest in processing *forecast queries* [4, 5], which is an approach to integrate forecast methods in standard relational query processing. In this context, a forecast query uses a *model* of the time series at hand to calculate the expected future behavior of the time series. Models used in [4] include sophisticated approaches from machine learning, like *support vector machines* (SVM) or *random forests* (RF), but traditional approaches from statistics, like the *autoregressive integrated moving average* (ARIMA) model, could also be used.

In any case, model-based forecasting basically involves two phases: *model creation* (often also called *training*) and *model usage*. The creation of forecast models is typically computationally expensive, often involving numerical optimization schemes to estimate model parameters. In addition, time series can be too large to fit in memory, so just a single scan over the series can be expensive [5]. If certain forecast queries are issued repeatedly, answer times of later queries can be reduced significantly if the model that was created during the first query is kept and maintained as necessary.

A good usage scenario for this approach is *data warehousing*. First, nearly every data warehouse can be considered as a large high-dimensional time series, since the time dimension is virtually guaranteed to be present [8]. Hence, the most basic requirement for forecasting, the existence of time series data, is met. Second, a significant number of repeated queries is issued, e.g. for creating reports. Therefore, it is safe to assume that a significant portion of the forecast queries issued to such a system could be answered with models that were created once and maintained thereafter. And most importantly, since data warehouses typically store measures that are used to make operational and strategic decisions, there is a natural and strong interest in applying forecast methods to this data [7].

In this scenario, the user does not want to state the suitable model instance for each forecast query explicitly. However, this is required in current extensions to RDBMS that integrate the creation and use of such models [11, 12]. Instead, users want to express their domain queries and expects the *transparent* use of applicable, existing models. This requires a mechanism to find these models quickly. We denote this task as *model matching*. Similarly, whenever new tuples are inserted (assuming append-only semantics), we need to find the models that are based on those. This means especially that the new tuples could be used to get more recent estimates of model parameters. We denote this task as *model maintenance*.

Both tasks seem similar to materialized view matching and maintenance. However, as we will explain in Section 4, the matching problem for forecast models differs from the matching problem for materialized views, as we can only use subsets. Regarding maintenance, forecast models only

DOI: https://doi.org/10.1145/1866480.1866485

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 26–31, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866485

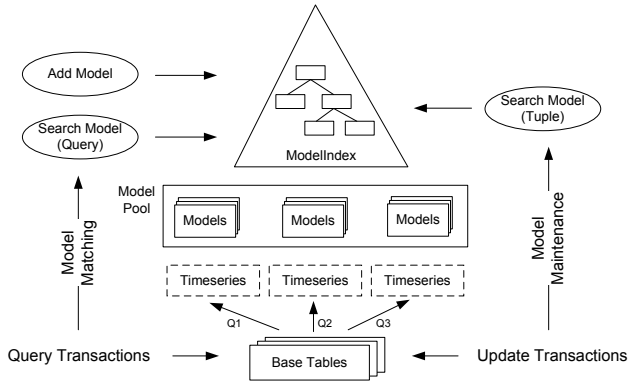


Figure 1: System Overview

materialize the parameters of a model, while a materialized view materializes data tuples.

Due to the possibly high number of individual time series in a data warehouse, the challenge is to efficiently find forecast models. Therefore, we propose a *model index* to index such models (Figure 1). Then, we show how this structure can be used for model matching and maintenance. Finally, we evaluate our model index with the TPCH benchmark and demonstrate significant improvement during query execution, but negligible overhead for update transations.

2. PROCESSING FORECAST QUERIES

In this section, we first define forecast queries and describe how these use existing forecast models transparently. We begin with the definition of time series relations that can be seen as a special kind of functional relations. [3]

DEFINITION 1. *Let S be a relation with schema $\{A_1, ..., A_m, y\}$ where $y \in \mathbb{R}$. The attributes $A_1, A_2...A_m$ are the* time *attributes of S. The attribute y is referred to as the* measure *attribute of S. Relation S is a* time series relation *(TR) if the dependency $A_1, A_2...A_m \rightarrow y$ holds. In addition, the following conditions must hold:*

1. *The relation is ordered on $\{A_1...A_m\}$, where $\{A_1...A_m\}$ form a time hierarchy (e.g. year, month).*
2. *There are no null values in S.*
3. *The elements of the composed set $\{A_1...A_m\}$ are unique and equidistant with respect to some application-dependent measure of distance.*

Examples for the measure of distance in the third condition might be month, day or weekday. To achieve equidistance, a preprocessing of the time series relation might be necessary (e.g. aggregation or adding missing values). Time series relations can be seen as logical views on base tables (Figure 1). Note that a star- or snowflake-schema is composed of one or several time series relations, since there is a functional dependency from dimensional attributes (including time) to the measure(s) in the fact table. Since such schemas are typically used in data warehousing, we focus on this application area in the rest of the paper.

We now define forecast queries in the setting of a data warehouse.

```
SELECT    country, YEAR(date) as
          year, MONTH(date) as
          month, SUM(salesunits)
FROM      facts f, regions r,
          products p
WHERE     f.city_id = r.city_id
AND       f.product_id =
          p.product_id
AND       p.pgroup IN ('phones',
          'netbooks')
GROUP BY country, year, month
FORECAST  6 months
```

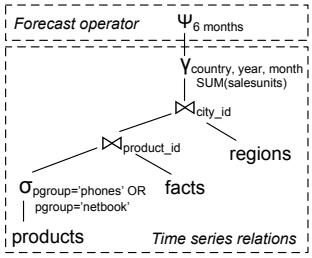


Figure 2: Forecast Query and Execution Plan

DEFINITION 2. *A forecast query defines one or several time series relations and a* forecast horizon. *The forecast horizon specifies the number of values to forecast or a future point in time until those forecast values will be required. A forecast query outputs* forecast values *for each time series relation according to the forecast horizon. The query has the following characteristics:*

1. *The query projects at least one time $|A_i| \geq 1$ and one measure $|y| \geq 1$ attribute.*
2. *The query can contain additional* nominal *attributes $\{C_1...C_n\}$. We call these* category attributes.
3. *The query is restricted to operations that fulfill conditions (2) and (3) of Definition 1. As a result, we only consider 1:N inner joins, so we do not have null values (no outer joins) or multiple assignments of one point in time (no N:M joins). Note that a typical data warehouse schema consists of foreign-key relationships anyway.*

We only consider univariate time series in this paper. Therefore, each measure attribute forms a separate time series relation together with each distinct set of values in the category columns.

EXAMPLE 1. *An example forecast query is shown in Figure 2 in the left part. Here, the user wants to forecast the sold quantities for products belonging to product group* `phones` *or* `netboooks` *for the next six months according to different countries. Measure (*`salesunits`*), time (*`date`*) and category (*`country`*) attributes are specified in the* `SELECT` *clause. With the* `FORECAST` *keyword the user specifies the forecast horizon. This SQL extension was introduced in [4].*

The corresponding plan (Figure 2 right) creates the time series relations (one for each country) in the first part and is extended with a relational *forecast operator* ψ on top.

This query will be processed as follows. The system needs to choose one or several *forecast models* from an existing *model pool* (Figure 1). The forecast plan operator is parameterized with the models to use and the forecast horizon. We assume the model pool is created by a decision analyst who knows which time series are queried and which models are appropriate for which time series. Each time series can be associated with several models. All models are indexed in a *model index*, which is used to find existing models (matching) and to add new models. If no model is found during the matching process (e.g. ad-hoc queries), we either return an empty result set or use a default forecast method. For example, we could use the automatic smoothing approach from

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 26–31, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866485

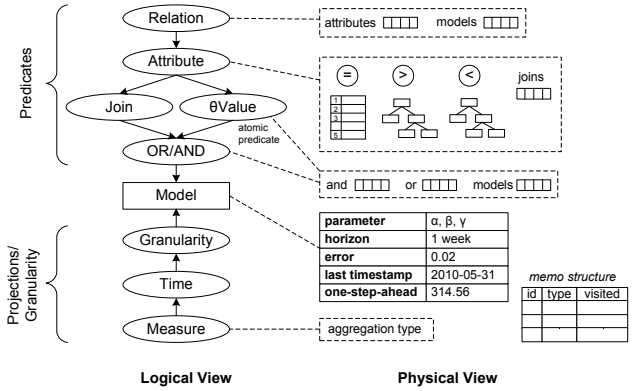


Figure 3: Model Index Structure

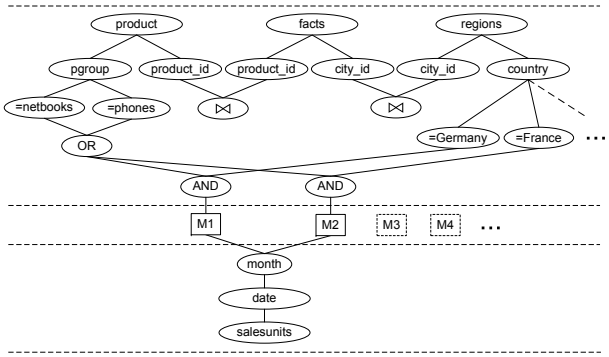


Figure 4: Example Model Index

Hyndman [7], which evaluates different exponential smoothing methods and chooses the best one. Then, the new model is added to the model index. For update transactions, the model index is used as well to find and maintain all models that are affected by the new tuple.

3. MODEL INDEX

The model index is a logical decision tree (Section 3.1), where different parts are modeled by different physical indexes (Section 3.2). The query definition of a time series is the key of the index, while the model itself is the value.

3.1 Logical Representation

A forecast model needs to be created for each time series relation that exists for each set of distinct values in the category columns and for each measure column. Therefore, for each time series, we can identify different parts: projection columns (measure, time), join paths, predicates (including value of category attribute) and time granularity. In Figure 3, the logical model index to index such a forecast query is shown in the left part. The model index consists of two parts. It can be entered from either part in order to retrieve the models that fulfill the conditions of that part. As we will explain later, for model maintenance only the upper part is required, while for matching both parts are used.

In the upper part, the predicates and join paths of the query are indexed. Our goal is to put the most selective conditions first in order to prune out non-matching models early. The predicates of a forecast query are normalized to the conjunctive normal form. Therefore, we first index atomic predicates of the form *attribute* θ v, where $\theta \in \{=, <, >\}$. We need to index the relation the attribute belongs to, the attribute itself, and the value v together with the condition θ. Atomic predicates can then be combined with disjunctions (OR node). Disjunctions can be combined with conjunctions (AND node). In addition, we need to express joins in the upper part. As we restrict join nodes to 1:N inner joins, we keep just a pointer from the join attribute of the source table to the join attribute of the joined table.

In the lower part, the projection columns and the time granularity are indexed. The entry points are the measure attributes, which point to time attributes. Time attributes can be distinguished by different time granularities (e.g., day, week). By storing several measure attributes, the model index can be easily extended for multivariate time series.

Every time a forecast model is created, the model definition is indexed in our model index structure. For example, Figure 4 shows the state of the model index after indexing models that are necessary for answering the forecast query in Example 1. As three tables are referenced in the query, we have three relation nodes. In the left corner, the inlist is indexed as disjunction, while in the right part, the column `country` is restricted. For each country, a forecast model has been created and indexed. The models are annotated with the used projection columns and time granularity *month* in the lower part of the index.

Note, the number of possible models is exponential to the number of attributes and distinct items. We can create models for all subsets over the domain of each attribute and for all subsets of attribute combinations.

3.2 Physical Representation

The physical realization of our model index is shown in Figure 3 in the right part. The model index is kept as a graph structure in main memory, as it needs to be accessed for every forecast query and every update transaction. Each part of the model index is only built as needed. All nodes can have multiple children and different types of child nodes. Conjunction and disjunction nodes have exactly two parents, while all other nodes have exactly one. Pointers to child nodes are stored in lists, hashmaps or b-trees. All nodes, besides attribute nodes, can point directly to models that fulfill the predicates up to that point. Therefore, relation nodes keep a list of attribute and model pointers. Attribute nodes keep a list of join nodes. The join node itself just keeps a pointer to the join attributes. Attribute nodes have two b-trees, one for greater- and one for less-than predicates. In addition, they keep a hashmap for equality predicates. This approach is similar to [10]. Atomic predicates ($\theta value$), conjunction and disjunction nodes have three lists – model, conjunction and disjunction pointers. Forecast models are the leaf nodes of the upper part of the model index. For every forecast model, we keep a model evaluation, the forecast horizon the model was optimized for, the last seen timestamp, and the one-step-ahead forecast in main memory. Most simple, but widely used, forecast models just require a few floating point numbers as parameters. For these models, we store the parameters in main memory as well. Otherwise, the parameters are materialized in a relational parameter table and we store the object identifier. In the lower part of the model index, we just store list of pointers to the corresponding child nodes. The measure node also

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 26–31, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866485

stores the aggregation type of the forecast query (e.g. sum, max). In addition, we keep a memo structure for conjunction and disjunction nodes (shown in the right corner). This memo structure is used for maintenance, as we will explain in Section 5. For maintenance as well, we keep a unique join index on each join column for all dimension tables.

4. MODEL MATCHING

In this section, we first study when a forecast model can be reused; then we will shortly discuss our matching algorithm.

The model matching problem can be expressed as follows.

DEFINITION 3. *Given a relational time series definition TR_Q, as defined in Definition 1, determine a possible set of models with time series definitions $TR_{M_1}...TR_{M_n}$ and an aggregation function AGG to calculate the forecast values of TR_Q. The following conditions must hold:*

1. *The attributes $\{A_1, ..., A_m, y\}$ of TR_Q and $TR_{M_1}...TR_{M_n}$ are identical.*
2. *The aggregation function AGG (e.g. sum) of TR_Q and $TR_{M_1}...TR_{M_n}$ is identical.*
3. *$TR_{M_1}...TR_{M_n}$ model the complete relation TR_Q by aggregation and optional duplicate elimination with regard to the aggregation function.*

Note, this definition also includes the exact match case where we reuse the exact suitable model for a query. In that case no aggregation is necessary.

Condition (3) makes the model matching problem more restrictive than the materialized view matching problem. To answer a query with an existing materialized view, it can also just be a superset of the query. Then, a selection predicate has to be applied. However, to answer a forecast query, we need to find all sets that represent the whole given time series relation. The problem of finding sets of time series relations is similar to the view-selection problem, which has shown to be NP-hard [1].

Therefore, we implemented only common cases in a data warehouse where reuse possibilities appear quite often. For example, we can reuse a model at lower time granularity and aggregate the forecast values. In our example (Figure 4), we can reuse the model $M1$ created for Germany to answer a query that wants to forecast the sum of the sold quantities for next year in Germany. We need to forecast twelve months with the existing model and then aggregate the forecast values, by using the specified aggregation method, to get a one-step-ahead forecast on year-granularity. A second case is the reuse of forecast models at lower dimensional hierarchies. Therefore, to get a forecast at a higher dimension, we need to forecast for all corresponding values at the lower dimension. For example, all models created in Figure 4 can be reused to answer a forecast query that forecasts total sales units for phones or netbooks for the next month. For each country, a forecast is calculated and all forecasts are aggregated to get the final result.

Note, although we can reuse models by aggregation of forecasts, the aggregated forecast might not be equal to the forecast directly created by a model. Several studies have analyzed this problem. Some concluded that aggregation of forecasts lead to better results [14], while others found little difference. Even so, we can always reuse an exact match without doubt, while aggregation should include an additional analysis of the forecast error.

Our model matching algorithm itself is called for each time series definition in a given query. It returns the node that fits exactly the query predicates. This could be a relation, atomic predicate, conjunction or disjunction node. If one or more models are directly associated with this node, we found an exact match. In the lower part of the model index, we need to check if measure and time column as well as time granularity fit. If so, we choose the model with the best model evaluation. If no exact match has been found, we check for a model at different time granularity. If no result has been found until now, we check for reusable subsets, where we use the set of models with the smallest number.

5. MODEL MAINTENANCE

Maintenance has to be done when the underlying base tables of forecast models change. Typical data warehouse applications are append-only, i.e. we only consider the insertion of tuples in the fact table that have a higher timestamp than existing tuples with the same dimension values. For maintenance, we traverse through the upper part of our model index for each newly inserted row in the fact table and identify all models that are affected by the current tuple. First, we need to maintain all models that are directly associated with the source table and have no predicates. Then, we need to check atomic predicates for the incoming tuple. For each single predicate that evaluates to true, we need to evaluate possible connections to other predicates. Note, for greater- and less-than predicates, a whole subtree might evaluate to true. To avoid visiting and possibly returning the same node multiple times, the search procedure must remember which nodes have been visited. Therefore, we use a recursive algorithm with memoization. Every time we find a new conjunction or disjunction node, we add its id to a memo structure (shown in the right corner in Figure 3). The memo structure consists of node ids, node types (conjunction or disjunction) and a flag indicating weather this node has been visited. For disjunction nodes, we evaluate the underlying subtree. When we enter a disjunction node a second time, we do not need to evaluate the subtree again. For conjunction nodes, we do nothing. When we enter a conjunction node a second time, we remove it from the memo structure and evaluate its subtree. We can do this since every conjunction node has exactly two parents. Every time we find a model as a child of a node, we maintain it. Finally, for each join node, we need to retrieve the corresponding tuple from the joined table in order to check additional predicates on this table. For this, we always have an index on each join column. Since we only consider 1:N joins, we can retrieve the joined tuple with a fast lookup. We now enter our index structure with the new tuple at the entry point of the dimension table.

For example, consider again the example model index in Figure 4. Assume, the following insert transaction is submitted `INSERT INTO facts(date, product_id, supplier_id, city_id, salesunits, purchaseunits) VALUES(2010-04-01, 1, 1, 34, 251, 953)`. As there are no predicates directly associated with the fact table, the maintenance search algorithm retrieves a tuple for `product_id=1` and `city_id=34` via the join index from `products` and `regions`. Assume `pgroup` equals `netbooks`, so the child *OR-node* of the *netbooks-node* and consecutively all child

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 26–31, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866485

AND-nodes are added to the memo structure. Then, when `country` equals `Germany`, the *AND-node* is found in the memo structure and therefore model $M1$ is maintained.

Finally, for each model found, the current model evaluation is updated with the new tuple. For this, we use the accuracy measure SMAPE (symmetric mean absolute percentage error), because of its scale-independence. Then, the model itself is maintained. Maintenance strongly depends on the used forecast method. However, after an insert transaction, we only want to perform cheap and incremental maintenance operations of forecasts and parameters. Therefore, each forecast method offers an interface called *IncrementalMaintenance*. For example, the parameters *slope* and *offset* of a simple linear regression model are calculated there, as this can be done incrementally. For an exponential smoothing model, only the next forecast value is calculated from the value inserted. However, if the model evaluation exceeds a certain user-set threshold ϕ, we invalidate it. Then, the next time the forecast model is used, we perform more expensive operations, so each forecast method also offers an interface *FullMaintenance*. For example, we could reestimate the parameters of a exponential smoothing model by rescanning the whole time series. With this approach, we let queries pay for maintenance, as it is certain that the result will be used.

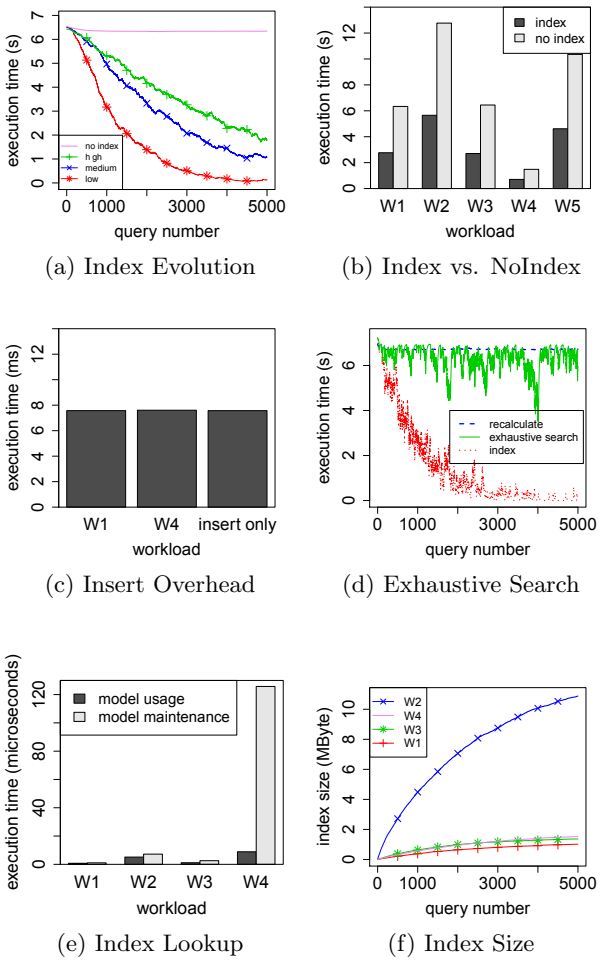
6. EXPERIMENTS

We implemented the described model index as a main-memory structure in PostgreSQL. Therefore, we extended the parser, optimizer and executor of PostgreSQL to support forecast queries. The parser annotates the time, measure and category attributes; the optimizer performs model matching, rewrites query plans if necessary and indexes new models. Finally, the executor builds new models if necessary or calculates forecasts from existing ones. In addition, we update existing models in the executor when new tuples are inserted into the system according to our maintenance algorithm. The test environment was an IBM Blade (Suse Linux, 64bit) with two processors (each a Dual Core Intel Xeon at 2.80 GHz) and 4 GB RAM.

We used data from the TPCH benchmark (scale factor two), which constitutes a normalized data warehouse schema, using primary and foreign keys between fact and dimension tables. We created five workloads. The first workload $W1$ forecasts the sold quantities according to different parts or suppliers. In the second workload $W2$, we use the same scenario but for several partkeys or suppliers using inlists. In workload $W3$, we forecast the sold quantities for different parts *and* suppliers. In the fourth workload $W4$, we forecast the sold quantities for different customers using a join with the orders table. Finally, in workload $W5$, we submit any combination of inlists, conjunctions and predicates used in the first four workloads. In order to conduct a fair evaluation, we use the simplest forecast method, *linear regression*, as default for all workloads. In the following experiments, we do not focus on forecast accuracy, as accuracy depends on the used data and forecast model, which is independent from our model index.

6.1 Model Usage

Figure 5(a)) shows the results of an experiment where we vary the number of distinct items (i.e. the size of the domain) in workload $W1$ and plot the execution times of



(a) Index Evolution (b) Index vs. NoIndex

(c) Insert Overhead (d) Exhaustive Search

(e) Index Lookup (f) Index Size

Figure 5: Experimental Results

5,000 queries. We set the number of distinct items to low (0.2 times the number of executed queries = 1,000), medium (0.5) and high (0.8). When we start the system with an empty model index, we need an initial phase where forecast models are built. With a low number of distinct items, the executed queries find a reusable model soon, so the average execution times decrease fast. We also plot execution times, when no index is used and forecast models are built for every new query. Due to caching effects, the execution times also decrease slightly in the beginning but stay constantly high until the end. Then, in Figure 5(b), we compare our index structure with the approach to use no index at all. We measure the average execution time of 5,000 queries and set the number of distinct items to *medium*. As a result, with our model index, we save about half the execution time for all workloads, as every second query finds a reusable model.

6.2 Model Maintenance

In a second series of experiments, we test the performance of our maintenance algorithm. First, in Figure 5(c), we show the total execution time of one insert transaction. For this, we create an initial model index of 5,000 models for workloads $W1$ and $W4$. Each insert requires the maintenance of about 100 models. In addition, for workload $W4$, each in-

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 26–31, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866485

sert requires an index lookup to dimension tables. We also show the execution time of a standard insert with no model index. As discussed in Section 5, each maintenance request on a model is very cheap, so the overhead is quite small. In a second experiment (Figure 5(d)), we submit queries and inserts according to workload $W1$ with a *low* number of distinct items. The ratio of queries to inserts is 99:1. We compare our approach with the no-index approach and an exhaustive search approach, where we just store model definitions in a list, kept in main memory. However, as we can only use it for exact matches, for maintenance, we drop all models that have been created for the concerned table. As a result, we get constant execution time when we use no index, and since we used a low number of distinct items, we get a strong decrease of execution times when we use our model index. With the exhaustive search approach, we need to drop all models every time a tuple is inserted into the lineitem table, so we result in fluctuating times.

6.3 Model Index Structure

Finally, we examine the model index more closely. Therefore, we indexed 1,000 models according to workload $W1 - W4$. Then, we measured just the lookup time in our index for model usage and model maintenance (Figure 5(e)). Note that all execution times are in the order of micro seconds, so the overhead is really small. The longest lookup time is found for workload $W4$, as we need to retrieve tuples from dimension tables. We also measured the lookup time of the exhaustive search list, explained in Subsection 6.2, which is around 87 micro seconds for all workload types (not shown). In a second experiment, we monitor the storage requirements of the model index for workload $W1 - W4$ and 5,000 queries. As we set the number of distinct items to medium, the size of the model index increases faster in the beginning than in the end. After 5,000 queries have been executed, the size of the model index is below 2 MByte for workloads $W1$, $W3$ and $W4$. For workload $W2$, we end up with a size of 10 MByte, as each inlist of size n requires the creation of $n - 1$ *OR-nodes*.

7. RELATED WORK

Forecast queries have been discussed in the context of the Fa system [4]. There, Duan and Babu proposed an incremental approach to build models in which more variables are added to the model in successive iterations. Another approach [5] addresses the issue of processing prediction queries over very large time series data sets. Forecasting is also supported as an extension of two common database systems. Oracle offers a `FORECAST` command as part of its OLAP DML [11]. The Data Mining Extension (DMX) in Microsoft SQL Server supports forecasting of time series using a custom forecast algorithm, which offers the interface method `PredictTimeSeries` [12]. However, all existing approaches lack the transparency of a seamless integration that does not require the selection of a model in a forecast query. In addition, they do not consider transparent maintenance. As our approach has shown negligible overhead in Section 6, it could be easily added on top of existing approaches to allow the transparent selection and maintenance of models.

Materialized views are similar to forecast models, as they need to be identified for maintenance and matching as well. However, to the best of our knowledge, no approach has developed a unique structure to use for both. Content-independent approaches [2] use rules to detect irrelevant updates to base relations. Content-dependent approaches use additional relations to check for irrelevant updates [13, 9]. All approaches filter out some materialized views, but they might still require checking against base relations. To speed up materialized view matching, a filter tree was proposed [6], which is an in-memory index and allows to find supersets or subsets of a given search key easily. However, as explained in Section 4, materialized view matching differs from model matching, as we only need to find complete sets.

8. CONCLUSION

We presented the novel concept of a model index that stores forecast models in main memory. We explained how this model index can be used to reuse and maintain existing forecast models. The proposed index is a tailor-made solution for forecast models. However, it could be used with certain extensions for any kind of statistical model as well.

9. REFERENCES

[1] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *PODS*, 1998.

[2] Jose A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[3] Héctor Corrada Bravo and Raghu Ramakrishnan. Optimizing mpf queries: decision support and probabilistic inference. In *SIGMOD*, 2007.

[4] Songyun Duan and Shivanath Babu. Processing forecasting queries. In *VLDB*, 2007.

[5] Tingjian Ge and Stan Zdonik. A skip-list approach for efficiently processing forecasting queries. *Proc. VLDB Endow.*, 1, 2008.

[6] Jonathan Goldstein and Per åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, 2001.

[7] Rob J. Hyndman, Anne B. Koehler, Ralph D. Snyder, and Simone Grose. A state space framework for automatic forecasting using exponential smoothing methods. *International Journal of Forecasting*, 18, 2000.

[8] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit*. Wiley, 2002.

[9] Gang Luo and Philip S. Yu. Content-based filtering for efficient online materialized view maintenance. In *CIKM*, 2008.

[10] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[11] Oracle. Oracle OLAP DML Reference: FORECAST - DML Statement, 2010.

[12] PredictTimeSeries – Microsoft SQL Server 2008 Books Online. `http://msdn.microsoft.com/en-us/library/ms132167.aspx`, 2010.

[13] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. 1996.

[14] David E. Rose. Forecasting aggregates of independent arima processes. *Journal of Econometrics*, 1977.