Combining Databases and Signal Processing in Plato [*]

Yannis Katsis
ikatsis@cs.ucsd.edu

Yoav Freund
yfreund@ucsd.edu

Yannis Papakonstantinou
yannis@cs.ucsd.edu

CSE Department, UC San Diego

ABSTRACT

Sensors generate large amounts of spatiotemporal data that have to be stored and analyzed. However, spatiotemporal data still lack the equivalent of a DBMS that would allow their declarative analysis. We argue that the reason for this is that DBMSs have been built with the assumption that the stored data are the ground truth. This is not the case with sensor measurements, which are merely incomplete and inaccurate samples of the ground truth. Based on this observation, we present *Plato*; an extensible DBMS for spatiotemporal sensor data that leverages signal processing algorithms to infer from the measurements the underlying ground truth in the form of statistical models. These models are then used to answer queries over the data. By operating on the model instead of the raw data, Plato achieves significant data compression and corresponding query processing speedup. Moreover, by employing models that separate the signal from the noise, Plato produces query results of higher quality than even the original measurements.

1. INTRODUCTION

Sensors generate ever increasing amounts of spatiotemporal data that have to be stored and analyzed. However, analysis of spatiotemporal sensor data is a labor-intensive process. In simple cases, the analyst copies (a subset of) the sensor measurements out of the data store to custom software (typically statistical signal processing algorithms), computes an underlying real world model, utilizes the model in various types of analyses, such as predictions, correlations and outlier detection and copies the results back into the database. In such cases, the database is utilized just as a store. In more complex cases, where the sensor analysis is combined with the context provided by the conventional alphanumeric data of the database, the analyst-specified processing pipeline is effectively a manually provided query plan.

By employing a hardcoded processing pipeline for each different analysis case, the state of the art in sensor data processing thus misses all the productivity and performance-enhancing features introduced by Database Management Systems (DBMSs), such as *declarative querying*, *automatic query optimization* and *independence of the physical layer* (specifying how data are stored and queried) *from the logical layer* (describing the view of the data to the analyst).

Though it is clear that a merger of signal processing and databases would be beneficial, conventional DBMSs cannot in their current form accomodate spatiotemporal signal analysis. The reason is a key difference in the founding assumptions of databases and signal processing. In contrast to conventional data found in databases, which have a one-to-one mapping to real world objects, the signal processing community assumes that the data are mere *measurements* (samples) of the corresponding physical reality. As such, sensor data are by definition (a) inaccurate, containing apart from the signal corresponding to the physical quantity measured also a noise component which is the result of random influences on the sensor and (b) incomplete, being discrete samples of a continuous signal. It is easy to see that conventional query processing fails under either of these properties. Querying noisy data leads to inaccurate results, without even explaining the extent of the inaccuracy. Similarly, even simple query operations, such as the join of two relations cannot be directly applied to incomplete data, since the two relations being joined may contain measurements that were taken at slightly different times and locations and thus do not align in space and/or time.

Plato aims to merge a general declarative DBMS (with all the associated advantages) with the signal processing paradigm, by operating on the underlying ground truth instead of the actual measurements. The ground truth in this case is represented by a *model*, which is a continuous function in space and/or time describing the quantity of interest (e.g., temperature, velocity, acceleration, air pollutant concentration etc.). Models have been successfully used in the signal processing community but have not yet been incorporated in DBMSs. The proposed Plato DBMS introduces models as first-class citizens. This addition not only enables declarative query processing for spatiotemporal sensor data, but also achieves significant compression compared to the storage of the actual measurements, which in turn leads to improved query processing performance. Lastly, using models that separate the signal from the noise leads also to improved accuracy of the query results.

In short, Plato proves the value of operating on the reality (as described by the models) rather than on its projection (as given by the raw measurements). This separation between the actual reality and the perceived reality is also the

[*]This work was partially supported by NSF awards IIS-1447943 and IIS-1237174.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA..

reason for the system's name, in an allusion to Plato's Cave allegory[1]. Plato proposes the following novelties:

- A DBMS architecture for spatiotemporal sensor data, containing a novel infrastructure for the definition of models through reusable learning algorithm modules. The models involve a probabilistic aspect that draws from the foundations of signal processing.

- Two query languages - ModelQL and InfinityQL - for declaratively querying models. Each language is suited to analysts of different backgrounds, exposing the models either as black boxes (best suited to statisticians and users of systems such as R) or as infinite relational tables (a good fit for SQL programmers).

- A novel hierarchical data store for models, in which the components of a model are stored in a compressed form in order of decreasing importance. This enables:

- A novel query processing algorithm operating directly on the models that brings significant gains in performance and accuracy compared to a naive solution operating on the measurements. The query processing algorithm leverages the hierarchical data store to further cut down query processing time by only utilizing the components of the model required to reach the confidence guarantees required by the user.

- Guidelines for inferring *reduced-noise* models; i.e., models that separate the noise from the signal and thus improve on the quality of the original measurements.

2. RELATED WORK

Models as tools for compression. Statistical models, primarily histograms and wavelets, have been widely used for compression in the database literature (see survey [1]). However, in contrast to Plato, these works assume the contents of the database to be the ground truth, rather than noisy and incomplete measurements thereof. Thus the query answering algorithms lack a probabilistic aspect. Furthermore, most works do not support arbitrary queries, being instead tuned to specific query patterns (e.g., approximate query result size through the use of histograms).

Model-based data infrastructure. The idea of using models as part of a general DBMS was first presented in the context of MauveDB [4, 2, 3]. Plato extends these ideas in several important directions: First, MauveDB argues that models need to be discretized in the coordinates' grid, before they can be queried. In this work we show that a fully virtual approach, where the model is perceived as a function over the infinite spatiotemporal domain, is both easier for the data analyst and more opportune for the query optimizer. For instance, consider two temporal models represented by their Fourier transform and a query asking for their correlation. It is most efficient to compute this query directly on the frequency domain rather than bring it back to the time domain. Second, while MauveDB showcases some of the challenges that arise in model-based systems and presents solutions for specific models, it does not contain a general framework that would allow one to plug in arbitrary models (which is one of the main goals of Plato). Lastly, other works studied particular point problems related to the use of models to represent sensor data, such as comparing compression ratios or designing indices useful for models [5, 8]. However, they did not present a general extensible model-based database platform.

Efficient query processing on models. The idea of evaluating queries directly on the representation of a model without discretizing them first, was presented in the context of FunctionDB [10]. The work showed that for a broad class of polynomial functions, faster processing is achieved by evaluating queries directly on the algebraic representation of the functions. Plato's goal is to provide the platform infrastructure that enables such optimizations for broader classes of statistical models.

Statistical functions inside DBMSs. Finally, libraries such as MADLib [7] allow statistical functions to be evaluated inside a DBMS. While this solves the problem of moving the data out of the database for processing, it couples model creation with query processing, thus not allowing the use of functions to generate models that are hidden from the analyst and used by the system to process arbitrary queries.

3. WHY MODELS?

Plato is based on the observation that sensor data can be stored and processed much more efficiently by operating not on the actual measurements but instead on a model of the underlying reality that can be inferred from the measurements. What is a model? In statistics a model is a statement about reality, which could be in general of any form. For instance, a model could be a fundamental law (e.g., Newton's law of acceleration, stating that the force vector $\vec{F}$ can be inferred from the acceleration vector $\vec{a}$ and the mass m through the formula $\vec{F} = m\vec{a}$) or a statement about a physical quantity in space and/or time in the past (e.g., the temperature at the entrance of the computer science department at UC San Diego on August 19, 2014 at 9am was between 68 and 70 degrees). But it could also be a predictive claim about the future (e.g., the average temperature of the earth will increase by at least 5 degrees between 2014 and 2050), or a theory (e.g., a medical theory stating that the statins decrease the chance of a heart attack by more than 20%).

Since we are interested in the processing of sensor data with a spatiotemporal component, in this work we restrict our focus to *quantitative spatiotemporal models*; i.e., models that provide the value of a physical quantity for points in space and/or time. An example of a quantitative spatiotemporal model is a function outputting the temperature at the entrance of the computer science department at different points in time. Signal processing has long recognized the value of operating on models inferred from the measurements, instead of operating on the actual measurements themselves.

Advantages of models. The reasons for preferring models over the raw measurements are multifold: Compared to raw sensor readings, models offer several advantages:

[1]In Plato's Cave, a few prisoners are bound, since birth, to look at a wall where shadows of real world's objects appear. The prisoners perceive only the 2-dimensional world of the shadows and miss the deeper insights that a comprehension of the full 3-dimensional reality would offer them.

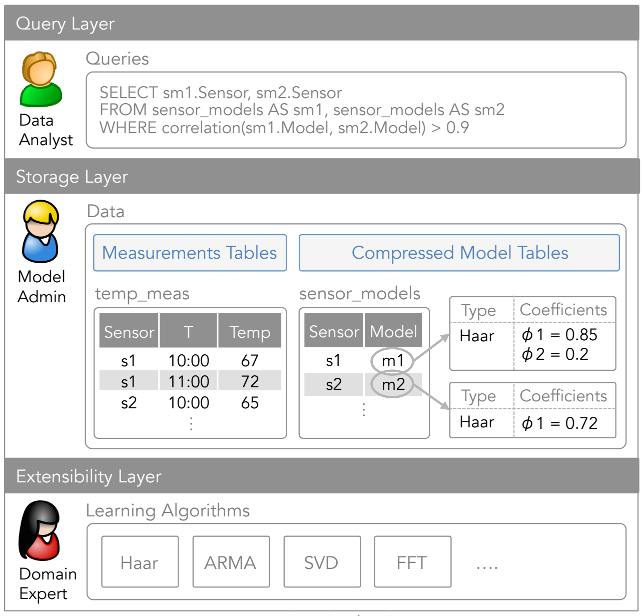
- *They are functions with possibly infinite domains.* Raw sensor readings are merely discrete samples of an underlying continuous phenomenon (i.e., they provide values of the measured quantity only for a finite number of points in space and/or time). In contrast, models provide also all intermediate values, intuitively "filling in" the gaps left by the raw measurements in the spatiotemporal dimensions. Having the values of the measured quantity for all points in space and/or time is especially important when joining two spatiotemporal signals on their spatiotemporal component, as otherwise the two signals may not be aligned. For instance, consider the following two datasets: A dataset containing air quality measurements taken at various locations and times at UC San Diego and another dataset containing GPS readings representing the location of a person walking around campus. If we want to compute the quality of the air the particular person was breathing during the walk, we would have to join these two datasets. However a conventional relational join on the space/time attributes of the two datasets will most probably yield the empty result, as the person may have never been at the exact time and location the air quality measurements were taken. Abstracting out each of these two sets of measurements through a corresponding model solves this problem and facilitates the join, as each model will provide the values for every point in space and time.

- *They offer predictive abilities.* In addition to providing values for points in space and/or time between those for which there exist raw measurements, models may also provide predictions for future points in time, thus allowing users to ask predictive queries. For instance, an air quality model may support queries about the expected air quality tomorrow. Intuitively, the predictive nature of the models stems from the fact that instead of focusing on the actual measurements, models instead capture the (typically recurring) pattern of the underlying phenomenon.

- *They improve accuracy.* By capturing the pattern of the underlying phenomenon, models may also be able to separate the noise (inevitably introduced in raw measurements due to the limited accuracy of sensors and other random factors) from the actual signal, leading to values that are more representative of the actual reality than the sensor measurements themselves. We will discuss in Section 5 how models may separate the noise from the signal returning values that are more accurate than the original raw measurements.

- *They capture uncertainty information.* Even if a model cannot completely separate the noise from the signal, it can explicitly capture the uncertainty that exists in the reported value for the measured quantity. This uncertain information is then leveraged by the query processing algorithms to generate query answers that themselves capture uncertainty. We will outline in Section 4.3 different ways in which a model can capture uncertainty and describe their relationship to existing works in uncertain and probabilistic databases.

- *They can be represented compactly.* Finally, models can be most of the time represented more compactly than raw measurements. This not only reduces the storage requirements for the - typically large - sensor datasets, but leads in many cases also to more efficient query execution, as queries can often be evaluated directly on the compressed model representation, as we will discuss in Section 6.



```
SELECT sm1.Sensor, sm2.Sensor
FROM sensor_models AS sm1, sensor_models AS sm2
WHERE correlation(sm1.Model, sm2.Model) > 0.9
```

Sensor	T	Temp
s1	10:00	67
s1	11:00	72
s2	10:00	65
⋮		

Sensor	Model
s1	m1
s2	m2
⋮	

Type	Coefficients
Haar	$\phi 1 = 0.85$ $\phi 2 = 0.2$

Type	Coefficients
Haar	$\phi 1 = 0.72$

Figure 1: Plato's Architecture

We next describe how Plato incorporates models and their associated advantages into a relational DBMS.

4. PLATO: A MODEL-AWARE DBMS

Plato enables declarative and efficient querying of spatiotemporal data through the architecture shown in Figure 1. The system interacts with three different types of users: *Domain experts* with knowledge of statistics and signal processing write learning algorithms, which given raw data create a model over the data using a particular statistical technique (e.g., Wavelets, Fast Fourier Transform, etc). Out of these registered learning algorithms, *model administrators* with knowledge of a particular dataset, choose a learning algorithm and instantiate its parameters to create a good model for the particular data set. Finally, *data analysts* query the generated models using a declarative query language. We next present the components of Plato enabling this workflow. As our running example, we will be using temperature data collected from sensors placed in offices across UC San Diego's campus, in the context of the Energy Dashboard project[2].

4.1 Preliminaries

In this work we consider sensor measurements with a spatial and/or temporal component. Let D_{xyzt} be the spatiotemporal domain that includes the 3D space and the time dimension and D_{desc} a subspace thereof. The subscript *desc* in D_{desc} describes the dimensions included in the subspace together with any range restrictions. For instance, $D_{xy:(x-x_0)^2+(y-y_0)^2=r^2}$ contains all 2D points within a circle centered at (x_0, y_0) with radius r. Finally, let X, Y,

[2]http://energy.ucsd.edu

Z, and T be the relational attributes corresponding to the three spatial and the temporal component, respectively.

Storing raw measurements. We consider sensor measurements that are stored together with their coordinates in conventional relational tables, which we refer to as *measurements tables*. A measurements table contains among others a subset of the spatiotemporal attributes X, Y, Z, and T together with a numerical attributes containing values of one or more measured quantities.

EXAMPLE 4.1. *For instance, table* `temp_meas(Sensor, T, Temp)` *of Figure 1 is a measurements table containing tuples of the form* (s, t, m)*, denoting that temperature sensor* s *provided the temperature measurement* m *at time* t. □

4.2 Deterministic Models

To enable declarative querying of spatiotemporal data, Plato allows the creation of models on top of the raw measurements. For now, we focus on deterministic models (i.e., models that do not contain any uncertainty), before showing how the concept of a model can be extended to capture the uncertainty inherent in sensor data. A deterministic model is intuitively a mathematical representation of the world that predicts a quantity of interest (e.g., temperature) at every point of the spatiotemporal domain. Formally:

DEFINITION 4.1. *A* deterministic model *is a function* $f : \mathcal{D} \mapsto \mathbb{R}$ *from a spatiotemporal domain* $\mathcal{D} \subseteq \mathcal{D}_{xyzt}$ *to the set* $\mathbb{R}$ *of real numbers.* [3]

EXAMPLE 4.2. *For instance, function* $f : \mathcal{D}_{t:2012 \leq t \leq 2013} \mapsto \mathbb{R}$ *is a model for temperature, which given a point in time* t *between 2012 and 2013 returns a real number corresponding to the* predicted *temperature at time* t. □

Generating models. Instead of writing models by hand, the model administrator creates models through *learning algorithms*, which take as input the measurements and potentially background knowledge about the real world and return a model fitting those measurements. The signal processing community has proposed a set of learning algorithms that have been shown to be applicable to a wide range of domains [9]. These include among others algorithms for learning ARMA (Autoregressive-moving-average) models (which are suitable for representing natural phenomena, such as temperature, where the present value depends on the recent past), SVD (Singular Value Decomposition) models, FFT (Fast Fourier Transform), Haar wavelets etc. Plato comes preloaded with several such learning algorithms and can be extended by domain experts with additional algorithms.

DEFINITION 4.2. *A* learning algorithm g *has the general form* $g(R; \bar{p})$*, taking as input a measurements table instance* R *and a (variable-length) tuple* $\bar{p}$ *of parameter values and returning a model.*

The variability of the length of $\bar{p}$ is crucial to ensure that the learning algorithm can be adapted to different scenarios (e.g., domains of different dimensionality).

[3]Although, for ease of exposition we restrict ourselves to models that return a single numeric value for each point in $\mathcal{D}$, in general a model could return values for more than one numerical attributes, i.e., a model could be a function $f : \mathcal{D} \mapsto \mathbb{R}^n$, $n \geq 1$.

EXAMPLE 4.3. *For instance, the learning algorithm* $haar(R; \langle \bar{A}_{cont}; A_{meas} \rangle)$ *takes as input a measurements table* R *together with the set* $\bar{A}_{cont}$ *of attributes of* R *that correspond to the spatiotemporal attributes and the attribute* A_{meas} *of* R *that corresponds to the measurement attribute of interest and creates a Haar model* $f : \mathcal{D} \mapsto \mathbb{R}$*, where* $\mathcal{D}$ *is the subspace of the spatiotemporal domain defined by attributes* $\bar{A}_{cont}$. □

Incorporating models into relational tables. To enable the seamless combination of models with standard relational data, Plato allows models to be used as values in tables. In addition to SQL's data types (e.g., string, integer, etc.), Plato also provides a new *model data type* that comes with an associated model signature $\mathcal{D} \mapsto \mathbb{R}$. An attribute of such a type is called a *model attribute* and accepts as values models conforming to the corresponding signature. We will refer to a table that contains at least one model attribute as a *model table*. Model tables are defined by SQL view definitions that involve *learning algorithm* invocations.

EXAMPLE 4.4. *The following statement creates out of the measurements table* `temp_meas` *the model table* `sensor_models`*, containing sensor IDs and Haar models, describing the predicted temperatures of the corresponding sensors:*

```
CREATE MATERIALIZED VIEW sensor_models AS
SELECT Sensor, haar(G; <T; Temp>) AS Model
FROM temp_meas GROUP BY Sensor AS G(T, Temp)
```

For ease of exposition, we use an extension of SQL that allows the creation of nested tables. In particular, the GROUP BY operator creates for each sensor in the measurements table `temp_meas` *a nested table G with all measurements for that sensor. This nested table is given as input to the Haar learning algorithm to create the corresponding model.* □

Similarly to conventional views, a model table may be virtual or materialized. In practice, the model administrator is motivated to materialize models (and the respective model tables) in order to benefit from the data compression that models enable, as we will discuss in Section 5. This can be achieved by using the MATERIALIZED keyword in the model table definition as shown above.

4.3 Probabilistic Models

For ease of exposition, models have been defined above as functions returning absolute values. However, since they are inherently statistical approximations of the underlying process (as they are based merely on measurements), models should have a probabilistic component. Adding probabilistic information to a model can be done in several ways. We next outline three alternatives, explain their connection to prior works in probabilistic databases and signal processing, and argue for the one we adopt in Plato. In the subsequent definitions we assume that the domain of the model is $\mathcal{D} \subseteq \mathcal{D}_{xyzt}$ and its intended range (leaving aside the probabilities for a moment) is $\mathbb{R}$.

Model as a probability distribution over functions. A *function-probability* model is a probability distribution over all functions $f : \mathcal{D} \mapsto \mathbb{R}$. Drawing an analogy with probabilistic databases, a function-probability model corresponds to a probabilistic database defined as a probability distribution over the set of database instances that constitute the set of possible worlds.[4] Although this definition of probabilistic databases is very general and thus guaranteed to cover any use cases, we are not aware of any practical system employing it. Similarly, we argue that function-probability models are merely of theoretical interest.

Model as a probability distribution over values. A *value-probability* model is a function from $\mathcal{D}$ to probability distributions over $\mathbb{R}$. Continuing our analogy, this corresponds to probabilistic databases where each tuple has a set of possible instantiations with associated probabilities that are independent of those of the other tuples. Since each point in $\mathcal{D}$ is assigned a probability distribution independently of the other points, value-probability models are strictly less expressive than function-probability models. However, value-probability models are still too general for practical use as they may attach an arbitrary probability distribution to a point in the domain $\mathcal{D}$. These probability distributions may be hard to infer from the data, hard to represent in a compact way and more importantly, they may contain more information than is really needed to reason about the data. Indeed, statisticians and practitioners often argue that knowing the exact probability distribution of the possible values for a point in $\mathcal{D}$ is not of practical importance. In order to make decisions based on the data, it suffices to know the range in which a value almost certainly will fall. This leads to the third definition of probabilistic models, inspired from statistics.

Model as a set of prediction intervals. A *prediction-interval* model is a function from $\mathcal{D}\times\mathcal{P}$ (where $\mathcal{P}$ the set of allowable p-values) to the set $\mathbb{R}_{interval}$ composed of triples of the form $(v, -\epsilon_1, +\epsilon_2)$, where $v, \epsilon_1, \epsilon_2 \in \mathbb{R}$. The semantics are the following: Let d be a value in the domain $\mathcal{D}$, p a p-value in $\mathcal{P}$ and $(v, -\epsilon_1, +\epsilon_2)$ the interval returned by the model for (d, p). Then, according to our current knowledge, the value corresponding to point d is in the interval $[v - \epsilon_1, v + \epsilon_2]$ with probability at least $1 - p$. The p-values are typically chosen from a set of values close to zero (commonly below 10%). Since the p-value p is close to zero, the probability $1 - p$ of the value being in the specified interval is close to one, making this an almost certainly true statement.

The inclusion only of statements that are almost certainly true is a fundamental departure from probabilistic databases, where one is interested in the probability of all possible events, however small that probability might be. Although general probabilistic databases have certainly their place in many scenarios (e.g., they are ideal candidates to store inferences made by bayesian networks), in this work we will adopt this restricted form of probabilities, as it has been proven in statistics to work well for decision support that leads to actionable items.

Note, that in addition to almost certainly true facts (i.e., facts that are true with probability at least $1 - p$), a model may also include almost certainly false facts (i.e., facts that are false with probability at least $1 - p$). This is essential in order to create a model that is closed under queries involving negation (which could start from a set of almost certainly true facts and infer a set of almost certainly false facts).

The signal-processing community has successfully used a special case of this model, briefly explained below.

Model as a combination of signal and noise. A *signal-noise* model is a special case of the *prediction-interval* model, where the intervals $(v, -\epsilon_1, +\epsilon_2)$ for each (d, p) pair in $\mathcal{D} \times \mathcal{P}$ are not given explicitly, but instead described indirectly through two components: (a) the center of the interval (i.e., the value v) at any point in $\mathcal{D}$ and (b) a gaussian distribution $N(0, 1)$ together with its amplitude at any point in $\mathcal{D}$, which are used to compute the range of the interval (i.e., the values $-\epsilon_1$ and $+\epsilon_2$) at any point in $\mathcal{D}$. The first component represents the *signal*, while the second represents the *noise*. In particular, for a given p-value a signal-noise model is a function $f(i) = s(i) + \alpha(i)n(i)$, $\forall i \in D$, where $s : \mathcal{D} \mapsto \mathbb{R}$ is a non-probabilistic function representing the signal, n is *white noise* (i.e., the well-known gaussian distribution $N(0, 1)$ [9]) and $\alpha : \mathcal{D} \mapsto \mathbb{R}$ is a function representing the amplitude of the noise at any point in $\mathcal{D}$.

Plato employs the prediction-interval probabilistic model and its signal-noise specialization as their value in capturing real use cases has been successfully proven by the signal processing community. Moreover, the latter approach allows models that separate the signal from the noise, leading to high quality data, as we will discuss in Section 5.

4.4 Queries

A key success factor of DBMSs has been declarative querying. Plato brings declarative querying to model tables through two declarative query languages aimed at two different classes of analysts. *ModelQL* exposes model attributes as black boxes on which statistical functions can be applied. This makes it perfect for analysts of a statistical background, who currently perform statistical analyses using statistical packages, such as R, SPSS, etc. *InfinityQL* on the other hand is best suited for SQL programmers that want to write standard SQL queries without having to worry about the existence of model attributes. To this end, *InfinityQL* exposes model attributes as nested relational tables with a conceptually infinite number of tuples. We next describe both query languages over non-probabilistic models, before showing how they can be extended to prediction-interval probabilistic models. Query processing is discussed in Section 6.

ModelQL: Querying models through functions. ModelQL allows analysts to query model attributes through model functions. A *model function* is a statistical function that takes as input a set $\bar{M}$ of models and returns a scalar, a tuple, a set of tuples, or a new model. For instance, a correlation function $cor(M1, M2)$ takes as input two models $M1$ and $M2$ and returns a scalar between 0 and 1 representing their correlation. Plato supports many common statistical functions and can be extended with additional functions. Given a set of model functions, a ModelQL query is defined as follows:

Definition 4.3. *A ModelQL query is any valid SQL query augmented with model functions that does not include in the projection list a model attribute.*

The restriction on the projection list guarantees that the

[4]In an even wider interpretation, a propability function characterizes the entire database and allows the expression of dependencies across different tuples and models.

result of any ModelQL query is a standard relational table.

EXAMPLE 4.5. *Continuing our running example, one can utilize the correlation function cir$(M1, M2)$ to compute all pairs of temperature sensors whose models have a strong (i.e., greater than 0.9) correlation as follows:*

```
SELECT sm1.Sensor, sm2.Sensor
FROM sensor_models AS sm1, sensor_models AS sm2
WHERE correlation(sm1.Model, sm2.Model) > 0.9  □
```

Although a perfect fit for statistical functions, ModelQL is not well-suited to relational operations, such as joins, selections, projections, etc. on models. Implementing each relational operator as a model function and writing a relational query as a composition of such functions is certainly possible in ModelQL. However, to offer a more suitable language for users coming from a SQL background, Plato offers a second query language for relational operations on model attributes, as described next.

InfinityQL: Querying models as infinite tables. InfinityQL allows analysts to query model tables by considering model attributes as nested tables with an infinite number of tuples. In particular, a model $f : \mathcal{D} \mapsto \mathbb{R}$ from vector (x, y, z, t) to r (conceptually) gives rise to an *infinite table* with schema $f(X, Y, Z, T, R)$, where (X, Y, Z, T) is the primary key. Intuitively, this table contains a tuple predicting the value of the quantities R for each value in $\mathcal{D}$. Given the definition of infinite tables, an InfinityQL query is defined as follows:

DEFINITION 4.4. *An InfinityQL query over a set of model tables is a nested SQL query over these tables, where model attributes are interpreted as nested infinite tables.*

EXAMPLE 4.6. *For instance, each Haar model in our running example can be seen as an infinite table over schema (T, Temp). Thus the model table* `sensor_models` *is conceptually a table over schema sensor_models$(sensor, (T, Temp))$, where $(T, Temp)$ is the schema of the nested table corresponding to the model. Using this representation, one can ask for the temperature of all sensors at midnight of 05/05/2012 through the following InfinityQL query:*

```
SELECT sm1.Sensor, m1.Temp
FROM sensor_models AS sm1, sm1.Model AS m1
WHERE T = 2012/05/05#00:00:00  □
```

Although InfinityQL queries may in general return infinite tables, to allow the visualization of query results, Plato supports only InfinityQL queries that are guaranteed to return a finite result. Such queries are called *safe*:

DEFINITION 4.5. *An InfinityQL query is* safe *if for every database instance it returns a finite result.*

Queries over probabilistic models. Both query languages can be extended to operate on prediction-interval probabilistic models. To adapt ModelQL and InfinityQL to probabilistic models, we make two revisions to the definitions presented above: First, we add a "WITH CONFIDENCE" clause that allows the analyst to specify the desired probability of the output being correct. Second, we change the output of the queries from tuples of simple values to tuples of values of the form $(v, -\epsilon_1, +\epsilon_2)$, such that each value v is guaranteed to be in the interval $[v - \epsilon_1, v + \epsilon_2]$ with confidence p.

EXAMPLE 4.7. *The query of Example 4.6 can be modified to return results with probability at least 0.95 as follows:*

```
SELECT sm1.Sensor, m1.Temp
FROM sensor_models AS sm1, sm1.Model AS m1
WHERE T = 2012/05/05#00:00:00
WITH CONFIDENCE 0.95  □
```

Given the desired confidence p of the query result, Plato's query processor automatically computes the p-value that should be used for each of the models that appear in the query's FROM clause in order to reach the desired confidence in the query's output.

5. COMPACT MODEL REPRESENTATION

As we discussed, models enable query processing on spatiotemporal sensor data. However, models also serve two other important roles: First, being succinct descriptions of the data, they can be represented compactly, leading to significant data compression. This leads not only to decreased storage requirements but also to improved query processing performance, since as we will see in Section 6, many queries can be evaluated directly on the storage representation. Second, by abstracting out from the specific data values, models can also separate the signal from the noise, leading to higher quality data compared to the original measurements.

Compression, noise and models are closely related. Good models allow Plato to achieve high compression ratios. In turn, high compression ratios indicate that Plato has correctly identified significant patterns in the data and removed the noise. We next review the state of the art in compression and describe the compression techniques employed in Plato.

5.1 State of the art in data compression

Compression methods are distinguished into lossless and lossy, depending on whether they retain or lose information from the input data, respectively.

Lossless compression. A lossless compression method (such as gzip and compress) can accurately reconstruct the original measurements. Looking at it as a statistical model over the data, its expected compression ratio is determined by the distance between the statistical model and the empirical distribution of the data (as measured by the Kullback-Leibler divergence). However, most of the time this compression ratio is around 2 to 4, as the lossless compression in order to accurately capture the entire input, models both the true state of the world (i.e., the signal) and any random influences on the measurements (i.e., the noise).

Lossy compression. In order to reach higher compression ratios, it is common to use *lossy compression*. Lossy compression is based on the assumption that the input data is a point in Euclidean space. For instance, let $\mathbf{x} = (x_1, x_2, \ldots, x_t)$ be a sequence of real values representing the measurements arriving from a (single) sensor. Similarly, let $\hat{\mathbf{x}} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_t)$ be the reconstruction of the sequence from the compressed version. If the compression is lossy, the difference $\mathbf{r} = \mathbf{x} - \hat{\mathbf{x}}$ (called the *residual*) is non-zero. For a lossy compression to be considered good, the size of the residual, also known as the *distortion*, should be small. The most common measure of distortion is the L_2 error, also called *root-mean-square-error* or *RMS*: $\mathrm{RMS}(\hat{\mathbf{x}}, \mathbf{x}) \doteq \sqrt{\frac{\sum_{i=1}^{t} r_i^2}{t}}$. Lossy compression methods such as jpeg2000 for images and mp3 for audio can achieve compression ratios of 100 or more with no perceptible degradation in quality.

5.2 Data compression in Plato

In this spectrum of compression options, Plato achieves a novel tradeoff: It achieves a high compression ratio (similar to lossy approaches), while simultaneously retaining sufficient information about the input data (similar to lossless approaches). Moreover, this information is arguably of higher quality than even the original measurements.

Reduced-noise models. This is achieved through *reduced-noise models*; i.e., models that separate the signal from the underlying noise. Instead of trying to minimize the amplitude of the residual (measured through the RMS), the learning algorithms employed by Plato make sure that the residual is white noise. For linear models one can check whether this is the case by considering the auto-correlation function for each residual and the cross-correlation between each pair of residuals. The residual is considered white noise when the auto-correlation consists of a single delta-function at zero and the cross-correlation functions are close to zero everywhere. Once the learning algorithm has successfully separated the signal from the noise, it creates a model of the form $f(i) = s(i) + \alpha(i)n(i)$, as described in Section 4.3, that retains the signal s, the description of the noise n (e.g., gaussian) and its amplitude α. By losing only the noise, the reduced-noise model is lossy (yielding a high compression ratio), but also retains high quality data (arguably exceeding the quality achieved by lossless models).

Improving models and compression by exploiting dependencies. The compression ratio can be further improved by building models that cover a set of proximate sensors, behaving similarly. For instance, consider our running example of temperature measurements taken from rooms in office buildings. Since neighboring rooms usually show similar temperature readings, it is beneficial for compression purposes to build a single model for all of them. Unfortunately, the space of models that capture all possible dependencies between sensors is extremely large and hard to learn fully automatically. Therefore, it is the role of the model administrator to specify the set of possible correlations that should be considered by the system to improve compression.

Accelerating query processing through additive models. Some of the most commonly used models for lossy compression, such as FFT, Wavelets, and SVD are composed of components that can be ordered according to their effect on the RMS measure. We call such models *additive*. Formally:

DEFINITION 5.1. *A model is said to be* additive *when it is a sum of components $\hat{\mathbf{x}} = \sum_{i=1}^{k} \mathbf{c}_i$ s.t. the following two conditions hold: (a) the best model for $k_2 > k_1$ shares the first k_1 components with the best model using k_1 components and (b) the reduction in the distortion is largest for the first component and decreases monotonically as k increases.*

Additive models lend themselves to an incremental compression scheme. Starting by setting the residual to the original measurements $\mathbf{r} \leftarrow \mathbf{x}$, we can create an additive model by performing the following procedure:

```
until $\mathbf{r}$ corresponds to white noise do
    Find the vector $\mathbf{c}$ that minimizes the RMS: $\mathrm{RMS}(\mathbf{c}, \mathbf{r})$
    Subtract the identified component from the residual:
        $\mathbf{r} = \mathbf{r} - \mathbf{c}$
```

By ordering the model components in decreasing order of importance, the query processing algorithm can perform incremental query answering, producing approximations of the result of ever increasing accuracy, as described next.

6. QUERY PROCESSING

In this section, we describe how queries are evaluated. For ease of exposition we use a variant of our running example involving a ModelQL query. However, similar ideas apply to the processing of InfinityQL queries.

EXAMPLE 6.1. *Consider a variant of our running example, where the temperature models are created through an FFT (Fast Fourier Transform) algorithm and stored as sets of frequency-amplitude pairs. In this setting, the following query asks for pairs of highly correlated temperature sensors.*

```
SELECT sm1.Sensor, sm2.Sensor
FROM sensor_models AS sm1, sensor_models AS sm2
WHERE correlation(sm1.Model, sm2.Model) > 0.9  □
```

To evaluate such a query, Plato offers two different query processing methods, depending on the information that is available about the functions involved in the query.

Processing queries by materializing model values on a grid. The baseline approach of processing queries is by materializing the values returned by each model on a spatiotemporal grid. This is similar to the approach proposed by MauveDB [4]. However, while in MauveDB the analyst has to manually specify the grid granularity, Plato automatically infers it, based on the query's desired confidence.

EXAMPLE 6.2. *The query of Example 6.1 is executed using the grid-based evaluation method as follows:*

```
SELECT sm1.Sensor, sm2.Sensor
FROM sensor_models AS sm1, sensor_models AS sm2
LET grid_start = min_coord(sm1.Model, sm2.Model)
LET grid_end = max_coord(sm1.Model, sm2.Model)
WHERE correlation(grid(sm1.Model, grid_start, grid_end,
60), grid(sm2.Model, grid_start, grid_end, 60)) > 0.9
```

where the function $\mathtt{grid}(f, l, u, s)$ reduces the model f to a discrete model f_d that is only defined on the grid specified by the start l, the end u and the step s. □

Processing queries directly on the model representations. Instead of discretizing models on the grid and subsequently applying the statistical functions, many functions can be evaluated directly on the storage representation of a model, therefore reaping the benefits of compression. For instance, the correlation function in Example 6.1 can be executed faster directly on the frequency representation.

To enable query processing directly on the storage representation of the models, the designer of a learning algorithm has to also provide corresponding implementations of the registered statistical functions. Plato's query processor automatically uses the appropriate implementation (e.g., correlation on the frequency domain), reverting to grid-based evaluation only when no suitable implementation is found.

Exploiting additive models. Evaluating the query directly on the model representations also allows the system to exploit the additive structure of the models. Given an additive model, Plato can compute the query result by first computing a rough approximation using the most important components and subsequently improving it by taking into account the remaining components. This enables the following three important features: (a) Improved query processing performance for queries with low to medium confidence, (b) Anytime query processing (i.e., answer queries with a strict deadline in an approximate manner) and (c) Online query processing, similar to online aggregation works [6], where a query returns a continuous result of ever increasing accuracy.

7. MODEL SELECTION

In Section 4.2 we saw how the model administrator can create models by employing learning algorithms that have been added to Plato. However, this assumes that the model administrator has a-priori knowledge of which type of model best fits the data to select the corresponding learning algorithm. For instance, in our running example this assumes that the administrator knew that a Haar model would be a good fit for temperature data. Although this assumption may hold in some cases (as the administrator may know that the particular phenomenon exhibits a periodicity that makes it a good candidate for Haar), in general, choosing the model that is the best fit for the raw measurements is a complex task that typically involves trying out different models and comparing them based on how well they fit the measurements.

Loss functions. The quantification of how well a model fits the data is typically done through *loss functions*. A loss function is a function that given as input a set of measurements and a corresponding model returns a non-negative real number, representing how well the model fits (i.e., predicts) the measurements. A low loss value indicates that the measurements *validate* the model, while a high loss value corresponds to *falsifying* the model by the data. The absolute value of the loss function is usually not important. It is the relative performance of each model w.r.t. the loss function that is important. Given a set of candidate models, their loss on the same data are compared and the best model (i.e., the model with the lowest loss) is chosen.

As we discussed in Section 5, the most commonly used loss function is RMS (root-mean-square-error). RMS is the square root of the sum of the squares of the difference between the predictions provided by the model and the actual outcomes. It is typically used when the model predicts real-valued quantities (e.g., the temperature as in our running example).

It is important to note however that other loss functions can be employed as well. The choice of loss function typically depends on the type of data described by the measurements. For instance, if the measurements are classifications of data into discrete categories, RMS is obviously not a very good metric of how good of a fit a model is w.r.t. those measurements. In this case, one usually employs as the loss function the fraction on mistakes (i.e., misclassifications) made by the model on unseen data.

The space of candidate models. Given a loss function suitable for the type of measurements at hand and a set of measurements, the model administrator wants to choose the model that best fits the measurements according to the particular loss function. To find such a model, the model administrator has to explore a variety of learning algorithms (potentially with different instantiations of their parameters), as they may lead to models of differing loss. For instance, Wavelets may be a better fit than FFT for a particular dataset (and vice versa).

Although this process is currently done manually, Plato has the potential of semi-automating it. In particular, if the space of candidate models is finite (which may not be the case, if for instance one has to consider an infinite amount of possible instantiations for the parameters of a learning algorithm), one could instruct Plato to automatically explore this space by creating all candidate models, and choosing the one with the lowest loss. We plan to investigate as part of our future work, whether this space of options is finite in practice and if this is the case, devise a language that allows the model administrator to compactly describe the space of candidate models that have to be explored.

8. CONCLUSION

As we have explained, combining signal processing and databases enables DBMSs that not only support spatiotemporal data but also leverage reduced-noise additive models to offer efficient and noise-free query processing. In the future, we plan to investigate whether models can be used not only as intermediate blocks enabling query processing but also as query results that provide insights into the structure of the data (e.g., correlations, dominant frequencies, etc.).

9. REFERENCES

[1] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2011.

[2] A. Deshpande, C. Guestrin, and S. Madden. Using Probabilistic Models for Data Management in Acquisitional Environments. In *CIDR*, 2005.

[3] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven Data Acquisition in Sensor Networks. In *VLDB*, 2004.

[4] A. Deshpande and S. Madden. MauveDB: Supporting Model-based User Views in Database Systems. In *SIGMOD*, pages 73–84, 2006.

[5] T. Guo, T. G. Papaioannou, and K. Aberer. Model-View Sensor Data Management in the Cloud. In *BigData Conference*, pages 282–290, 2013.

[6] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD*, pages 171–182, 1997.

[7] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB*, 5(12), 2012.

[8] N. Q. V. Hung, H. Jeung, and K. Aberer. An Evaluation of Model-Based Approaches to Sensor Data Compression. *IEEE TKDE*, 25(11):2434–2447, 2013.

[9] J. Stein. *Digital Signal Processing: A Computer Science Perspective*. Wiley-Interscience, 2000.

[10] A. Thiagarajan and S. Madden. Querying Continuous Functions in a Database System. In *SIGMOD*, 2008.