The healthy aspects of SQL databases in analytics Since the 80s SQL database systems gradually became the cornerstone of the vast majority of Business Intelligence (BI) and On Line Analytical Processing (OLAP) applications. Their success is primarily attributed to *declarative querying (via SQL)*, *automatic query optimization* and the *physical-logical independence*. In a typical database-driven application architecture (see Figure 1), the business intelligence application issues an SQL query over tables. The tables are *logical* in the sense that they are just mathematical relations, while the underlying storage, clustering and indexing details constitute a *physical* layer, whose knowledge is not needed for writing queries. The queries are declarative in the sense that they only describe the desired result without describing the algorithm that computes the result. A query optimizer automatically decides which is the best plan for executing the query, taking into consideration the physical organization of the storage. A database administrator may alter the data structures and indices from time to time in order boost the performance of the observed workload. However, logical/physical separation dictates that fine-tuning changes at the "physical" level do not require rewriting of the logical layer. Rather, the query processing speed will automatically benefit.
[[[to Yannis P, introduce the following at the point where models emerge as added-value views: Further levels of abstraction are achieved with *views*, which are tables that capture the result of filtering, combining and aggregating the data of the logical tables in a way that simplifies the queries issued by the applications.]]] <==

The failures of SQL databases in spatiotemporal sensor data analytics SQL systems fail in the space of predictive analytics involving sensors that generate measurements in continuous space and time [1]
Yoav: *In this context it is important to distinguish between the* real world model *and the* sensor model*. The real world model is a set of functions which map a* continuous coordinate system *(such as time and space) to quantities such as temperature or air quality. The sensor model defines the location of the sensors in the real world model, the times in which they sample, and the value sequences collected from the sensors. The tension comes from the fact that the only source of data: the measurements, are noisy and discrete in time and space, while the real world model is defined for all coordinate values.*

Next, we argue that at the core of the failures of SQL databases is the fact that they lack a critical abstraction, which is the *Real World Model*.[2] Then we argue next that this omission leads to multiple failures: Tedious programming and system integration, lack of abstractions, which requires developers/analysts to jungle the high level, where they think of statistics, such as the correlation of two sequences, with the low level programming where they deal with the specifics of the model. The second failure is that often the massive amounts of measurements are used directly in the statistics, leading to a performance waste.
In general, the data contained in an SQL database have a one-to-one mapping to the captured real world objects and events. For example, a database may represent persons, which have an address, a name and a telephone number. Each represented person is supposed to be a corresponding unique entity in the world. Vice versa, as far as the database and the applications running on it are concerned, the represented entities, attributes and relationships constitute the entire real world of interest. On the other hand, the sensor data are mere samples (measurements) of the reality. The full reality of interest to the applications is captured by models that predict the quantity of interest at each point in a range of time and possibly also in an area of space. In practice, the gap between the measurements and the real world models is best crossed by learning algorithms that discover the parameters of the real world model.
While databases are often used to store sensor measurements, their deficiency in capturing real world models as a first class citizen leads to tedious analytics processes: Given an analytics request, one has to first issue queries that retrieve the relevant measurements. Then she deploys signal processing learning algorithms on them to compute the model's parameters. Finally, the (typically statistical) analysis is often performed over the model and its results are moved into the database.
In recent years databases have been extended (typically via UDFs) with machine learning procedures that

[1] Indeed, one may argue that some of the discussed failures and respective remedies proposed by Plato, also apply to predictive analytics of non-sensor, non-spatiotemporal data. Nevertheless, the focus of this project is exclusively on spatiotemporal sensor data.
[2] The real world model is frequently called the "physical model" as it corresponds to the physics of the real world. This proposal uses the term "real world model" (instead of the term "physical model") to avoid the confusion caused by the two different meanings of the word "physical" in the sensors' signal processing and the database community.

the conventional database-driven analytics architecture illustrating declarative query and optimizer that automatically finds how to utilize the data structures best.

Figure 1: Database-driven analytics architecture

can allow one to execute such procedures without moving data in-and-out of the database. While such extensions are useful, we argue next that they miss the optimization opportunities that become available once models appear in the database as first class citizens and the query language and query processing are adjusted accordingly.

The net effect is that analytics on spatiotemporal sensor data currently require the involvement of developers who are essentially skilled in all of the (a) data filtering and combining, (b) modeling, and (c) low level data representation and related algorithms.

[[[REMINDERS ON POINTS FOR INTRODUCTION]]] <==

Problems of current approaches to combining databases with statistical signal processing

- (Done) Statistics and predictive analytics involve a tedious process where data are moved out of databases into statistical processing software (eg R) and then stored back.
- None of the cornerstones of the success of SQL systems is available: No logical/physical separation, no declarative queries and no automatic optimization of storage and query processing.
- (Done) Spatiotemporal sensor data are qualitatively different in a way that prevents the direct use of SQL, which has been tuned for discrete precise data: Unlike ERP and CRM business use cases, where the database data constitute the world, sensor data are mere measurements of the real world - potentially imprecise. Models (continuous functions over space and time) are the actual representation of the world and predict values at any coordinate. Query languages and database modeling has to facilitate the use of predicted values.
- Sensor data are unnecessarilly big in their raw form. Yet signal processing research has known for long that data can be split into signal and the often unimportant noise, leading to high effective lossless compression.

The benefits of a model-aware database

- Plato brings models in the database. Models are continous functions that predict the quantity of interest at any coordinate. A model introduces knowledge of the external physical reality.
- Well-developed theory and set of models and model generators that effectively capture many real world cases.
- Holds promise to solve the problems above. Plato enables declarative queries and automatic optimization of storage and query processing.

Research Challenges

- Specify declarative query languages that process models.
- The database algorithms can work efficiently directly on the models. Specify a logically/physically-separated architecture where an optimizer is aware of the specifics of the model representation and chooses query processing algorithms accordingly. For example, consider two models represented by their Fourier transform and a query that asks for their correlation. It is most efficient to compute directly on the frequency domain rather than bringing back to time domain.
- Signal processing community provides a wide variety of model generators, with various guarantees on the loss information. Plato must semiautomate the choice of the appropriate model generator. This requires a classification of model generators with respect to their loss of information properties. More importantly, different queries may pose different needs of accuracy and of what constitues information loss. The semiautomation must take into account query workload. The query answering must utilize the best model for the problem. Granularity of queries can also play a role: Can you avoid computing the entire model and instead compute on the fly only the parts of the model that are of interest to the query? Which models are best fo such?
- How do you incrementally maintain the model as the data change? Tradeoff between speed of convergence and computational resources used. Again, query workload must specify.

Use cases

0.1 Prior work on model-aware databases

[[[to Yannis K: Please check and add to the representations I make about the prior work.]]] <==

Works from ML and statistical signal processing communities [[[for Yoav]]] <==

MauveDB Argued for the value of direct SQL processing. MauveDB uses models to predict values on a developer-specified grid.

- Plato argues that models need no be discretized in the coordinates' grid. A fully virtual approach, where the model is perceived as a continous function, is easier for the developer and more opportune for the optimizer. For example, consider two models represented by their Fourier transform and a query that asks for their correlation. It is most efficient to compute directly on the frequency domain rather than bringing back to time domain.
- Did not investigate the connections between (a) choosing models and query workload and (b) query guarantees given chosen models.

Function Db (MIT)

- Showed that in the case of models captured by polynomials better compression and faster processing is achieved by working directly on polynomials.

Aberer [[[for Yannis K: It seems he promised a lot and did a little. What do we write about this?]]] <==

Approximate query processing

1 Models

[[[To Yannis P: I changed $\bar{\mathcal{D}}$ to $\mathcal{D}$, due to the reasons we discussed. Similarly for $\bar{\mathcal{Q}}$.]]] <==

The proposed Plato database aims to solve the discussed problems by introducing models as first class citizens. In this section we formally define models and discuss the criteria that can be used to compare two models.

A model is a mathematical representation of the world that predicts a quantity of interest (e.g., temperature) at every point of the spatiotemporal domain. In the basic case, a *model* is a continuous function $f : \mathcal{D} \mapsto \mathcal{Q}$ from a (generally multidimensional) spatiotemporal domain $\mathcal{D}$ to $\mathcal{Q}$. In this work we consider function domains $\mathcal{D}$ that are subspaces of four-dimensional spatiotemporal domain (that includes the three space dimensions and the temporal dimension). For instance, it can be a sphere of (x, y, t) points around a center (x_0, y_0, t_0), a square of (x, y) points, etc.

EXAMPLE 1.1 For instance, a model for temperature can be a function $f : D_{t\_2012\_now} \mapsto Q_{float}$, which given a point in time t between 2012 and now returns a float, corresponding to the *predicted* temperature at time t. □

2 Plato: An extensible model-aware database

Platon facilitates the integration of models in a general database system by adopting the architecture shown in Figure **??** and discussed next.

Raw sensor data measurements are stored in conventional SQL tables, which we refer to as *measurements tables*. Every measurements table contains among others a subset of the spatiotemporal attributes X, Y, Z and T (representing the three spatial dimensions and the temporal dimension, respectively).

EXAMPLE 2.1 Consider a measurements table with schema `temp_meas(Sensor, T, Temp)` containing tuple of the form (s, t, m), signifying that temperature sensor s provided at time t the temperature measurement m. Attribute `Sensor` is a foreign key to another table `temp_sensor(ID, X, Y, Installed_on, ...)` providing the properties of each sensor, including among others its latitude and longitude.[3] □

To facilitate easy data analysis, the model administrator can build models on top of the raw sensor data. To generate a model, the model administrator can utilize *learning algorithms*, which take as input the raw data

[3] The data administrator can decide what is the best schema for the measurements data. For example, instead of a single table carrying all the temperature sensor measurements, the administrator may choose one table for each sensor.

and potentially external knowledge about them and return a model. Learning algorithms are designed by domain specialists, who have knowledge of the domain and the underlying physical properties of the sensor data. For instance, weather experts can design a weather learning algorithm. Although some of the learning algorithms will be domain specific (such as the weather example above), others are general algorithms proposed in the machine learning literature [[[Yoav, can you please add a reference?]]] and shown to be applicable to a wide range of domains. This includes among others algorithms for learning an ARMA mode (which is suitable for readings from sensors measuring natural phenomena, such as temperature), a PCA model (Principal Component Analysis) etc. To bootstrap the system and facilitate the quick generation of models for many common use cases, Plato comes preloaded with several such learning algorithms.
A learning algorithm g has the general form of $g(R; p)$, taking as input a measurements table R and a tuple p of parameter values. <==

EXAMPLE 2.2 For instance, the learning algorithm $arma(R; < Attr_{cont}; Attr_{meas} >)$ takes as input a measurements table R together with the sets $Attr_{cont}$, $Attr_{meas}$ of attributes of R that correspond to the spatiotemporal attributes and measurement attributes of R, respectively and creates an ARMA model $f : \mathcal{D} \mapsto \mathcal{Q}$, where $\mathcal{D}$ is the subspace of the spatiotemporal domain defined by attributes $Attr_{cont}$ and $\mathcal{Q}$ is the equal to $Q_1 \times Q_2 \ldots \times Q_n$, where Q_i the domain of the i-th attribute in $Attr_{meas}$. [[[Which additional parameters should we include to the ARMA learning algorithm's signature to make it generally applicable?]]] <== □

Models created through learning algorithms can be used as values in tables. To this end, Plato extends SQL's data types (e.g., string, integer, etc.), with a new *model data type* that comes with an associated model signature $\mathcal{D} \mapsto \mathcal{Q}$. An attribute of such a type is called a *model attribute* and accepts as values models conforming to the corresponding function 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 2.3 For instance, in order to create a model table, containing sensor IDs and an ARMA model, representing the predicted temperatures for the particular sensor, one can write the following statement:

```
CREATE MATERIALIZED VIEW sensor_models AS
SELECT sensor, arma(G; T; Temp) AS model
FROM temp_meas GROUP BY sensor AS G(T, Temp)
```

Note that for ease of exposition, we use an extension of SQL that allows the generation of nested relational tables.[4] In particular the GROUP BY operator creates for each sensor appearing the the measurements table `temp_meas` a nested table G containing all measurements for the particular sensor. These measurements are given as input to the ARMA learning algorithm to create the corresponding ARMA model. The resulting model table `sensor_models(sensor, model)` contains tuples of the form (s, f), where s is a sensor and f the corresponding temperature ARMA model. □

A model table may be virtual in the sense that SQL view definitions are virtual, i.e., no actual computation is performed until a query uses them. However, in practice, the administrator is motivated to materialize models (and the respective model tables) in order to benefit from the data compression that models enable. This can be achieved by adding the MATERIALIZED keyword in the view definition as shown above.
Note, that the administrator has in general to choose in general between multiple model algorithms, which come with different compression levels and corresponding guarantees regarding information preservation. Indeed, a major research issue, discussed in Section **??**, is the semiautomation of the choice of model algorithms, which should take into consideration the query/analytics workload.

[4] Extending SQL to nested tables has been a well-studied topic in the database management field and recently it is featured in multiple "Big Data" databases that feature semistructured models.

EXAMPLE 2.4 In a modification of the running example, the administrator can choose to capture temperature in a single model `full_model` that predicts the temperature based on `x`, `y` and `t` and is defined as follows:

```
CREATE MATERIALIZED VIEW full_model AS
arma(SELECT X, Y, T, Temp
FROM temp_meas, temp_sensor WHERE Sensor = ID; X, Y, T; Temp)
```

The reasons for doing so could be: (a) The individual sensor models may be heavily correlated based on the `X` and `Y` sensor coordinates. In such case, a single model can have a much more compact representation than the collection of individual sensor models. (b) The analytics may need temperature predictions for specific locations, which do not coincide with any individual sensor. □

Probabilistic model tables In the general case a model returns probability distributions of the quantities, rather than absolute values. Therefore a model is a function $f : \mathcal{D} \mapsto \mathcal{H}_{\mathcal{Q}}$, where $\mathcal{H}_{\mathcal{Q}}$ is the space of probability distributions over the quantity domain $\mathcal{Q}$. The probability distributions capture the certainty of the predicted values. For example, the model of Example 2.3 can easily benefit from returning probability distributions of the predicted temperatures. A key use case is that the probability distributions indicate how certain one is about the confidence intervals of the predicted quantities.

2.1 Queries and Optimized Query Evaluation

A key success factor of database systems has been the declarative SQL query language where the user/developer expresses the desired result of his analysis without having to specify the algorithm that computes this result and without having to refer to the data structures where the data are stored. The database discovers the optimal plan to compute the result, making best use of the available data structures.

In the spirit of declarative programming, Plato offers an extension of SQL that enables computations involving models. A data analyst can query models in two ways, corresponding to queries of different granularities. At the coarsest granularity, he can query models as black boxes by employing in the query predefined functions that take as input one or more models and return a scalar, vector or even a new model. Examples of such functions are functions computing statistical properties of models, such as the correlation between a model or the autocorrelation within a model. Plato will support many common such functions, while additional functions can be added through a suitable interface.

EXAMPLE 2.5 The function $correlation(M1, M2)$ takes as input two models $M1$ and $M2$ and computes their correlation. Continuing our running example, one can utilize this function to compute all pairs of 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
```

□

[[[Need to motivate the split into two query language extensions]]] An alternative way of accessing the models is by considering them as nested tables and using an extension of SQL suited for querying nested relational models. In particular, a model $f : \mathcal{D} \mapsto \mathcal{Q}$ from vector (x, y, z, t) to vector $(q_1, q_2, \ldots q_n)$ can be seen as a table of schema $f(X, Y, Z, T, Q_1, Q_2, \ldots, Q_n)$, where (X, Y, Z, T) is a primary key. Moreover, conceptually contains a single tuple for each value in $\mathcal{D}$. [[[Need to fix the transition from domains to attributes.]]] Since the latter is in general infinite, the model is also exposed to the data analyst as an infinite relation. This infinite relation can be queried through SQL queries (extended to account for nested relations). <== <==

EXAMPLE 2.6 For instance, each ARMA model in our running example can be seen as a table over schema (T, Temp). Thus the model table `sensor_models` defined above is a table over schema $sensor\_models(sensor, (T, Ten$ where $(T, Temp)$ is the schema of the nested table corresponding to the model. Using this representation, we can ask for the temperature of all sensors at midnight of 01/01/2014 through the following query:

```
SELECT sm1.sensor, m1.Temp
FROM sensor_models AS sm1, sm1.model AS m1
WHERE T = 2014/01/01#00:00:00
```
□

[[[Add discussion on safe queries]]] <==

Unlike conventional query optimization, where the rewritings produce equivalent expressions, opportune rewritings in model-based databases are not guaranteed to produce queries with identical results. Rather, in many important cases the results are guaranteed to be equivalent under common assumptions of statistical signal processing. In other cases, they are guaranteed to be equivalent within certain error bounds. Plato queries allow the user to specify the requested guarantee by providing appropriate parameters.

3 Opportunities in Data Compression

As is well known from signal processing, relatively few model classes, such as ARMA models, Fourier and wavelet-based models and Singular Value Decomposition (SVD) based models capture very well various scenarios. The key function of the raw data administrator is to choose the appropriate *model generator* and feed it with the appropriate parameters that will dictate compression, accuracy. The important lesson from signal processing is that high compression can be achieved with minimal or even no loss of accuracy.

3.1 Choosing model alternatives

[[[To Yoav: Please add discussion on the following: (a) Lossless: Does not achieve significant compression, (b) Noise-reducted lossy, (c) Additive hierarchy of (b)]]] <==

Yoav: *From a mathematical point of view, there is a close connection between fitting a probabilistic model to data and compressing the data. A both are central to the workings of Plato, we briefly describe them here.*

3.1.1 Lossless compression

A lossless compression method is one which can reconstruct the original measurements without error. A lossless compression method corresponds to an assumption about the statistical model over the data and. The compression ratio is a function of the divergence (the Kullback-Leibler-divergence) between the assumed distribution and the true distribution. Thus the compression ratio is a measure of the randomness in the sensor data. Unfortunately, lossless compression methods usually achieve a compression ratio of around 2-4. Which corresponds to the fact that they accurately capture both the true state of the world (the signal) and the many random influences on the measurements (the noise). Lossless compression cannot distinguish between the two.

3.1.2 Lossy compression

In order to reach higher compression ratios than lossless compression. It is common to use Lossy compression *[?]. In this case, instead of requiring perfect reconstruction of the original measurements, we allow a quantifyable amount of* distortion *in the reconstruction. Suppose* $\langle x_1, \ldots, x_t \rangle$ *is the original sequence and the reconstructed sequence is* $\langle \hat{x}_1, \ldots, \hat{x}_t \rangle$*. By far, the most common measure of distortion is the* root-mean-square *or RMS:*

$$RMS(\langle x_1, \ldots, x_t \rangle, \hat{x}_1, \ldots, \hat{x}_t) \doteq \sqrt{\frac{\sum_{i=1}^{t} (x_i = \hat{x}_i)^2}{t}}$$

If the errors are usually small and only rarly very large, the RMS is misleadingly large. A better error measure in such situations is the fraction of times the error is larger than some threshold:

$$err_\epsilon(\langle x_1, \ldots, x_t \rangle, \hat{x}_1, \ldots, \hat{x}_t) \doteq \frac{\sum_{i=1}^{t} \mathbf{1}(|x_i - \hat{x}_i| > \epsilon)}{t}$$

Lossy compression methods such as `jpeg2000` *for images and* `mp3` *for audio can achieve compression ratios of 100 and more with no perceptible degredation in quality.*

3.1.3 Noise reduction through compression

Lossy compression methods invariably lose some of the original information. However, it is often the case that the result is a reconstruction that is closer *to the true underlying real-world values than the original measurements. How can that be?*

Consider a temperature guage in a room that is taking a measurement ten times per second. Suppose also that each measurement is the sum of the true temperature and measurement noise with a variance of one degree. It is clear that replacing blocks of 10 consecutive measurements by their average is lossy in terms of the original signal. However, at thge same time, it is closer to the true measurements. It is not uncommon that compression methods that incorporate good model of the underlying real-world process can actually clean *the raw measurements.*

3.1.4 Incremental models

A powerful way of building compression methods is based on iterative compression of the residual. Using the notation above, the residual is $r_i = x_i - \hat{x}_i$. *We expect the resudual to be small.*

3.2 Adjusting to filtering needs

[[[To Yannis K: Add discussion explaining which models are good for pushing selections down]]] <==

4 Query Processing

EXAMPLE 4.1 The following query discovers the pairs of highly correlated temperature sensors and will be the running example of this section.

```
SELECT sm1.sensor, sm2.sensor
FROM sensor_models sm1, sensor_models sm2
WHERE corelation(sm1.model, sm2.model) > 0.9
```

[[[to Yoav (Priority 2): An obvious and inefficient way to find the required pairs is to try all pairs. However, there is a better way, based on the transitivity of correlation. Eg, if corelation(x,y) is very close to 1 and corelation(y,z) is very close to 1 it should be that correlation(x,z) is somewhat close to 1. Let's discuss it because there may be good space for rewriting optimizations, i.e., showing how an optimizer can capture such patterns and execute in more efficient plans.]]] <== □

Processing queries by materializing model values on a grid MauveDB suggested that the analyst specifies a spatiotemporal grid and each model participating in a query returns the quantities at the grid's points. Consequently, the statistical functions of the query can be computed in a straightforward way.

EXAMPLE 4.2 The following Plato query utilizes MauveDB's technique. It dictates a grid-based execution of the query of Example 4.1.

```
SELECT sm1.sensor, sm2.sensor
FROM sensor_models sm1, sensor_models sm2
LET grid_start = min_coord(sm1.model, sm2.model)
LET grid_end = max_coord(sm1.model, sm2.model)
WHERE corelation(grid(sm1.model, grid_start, grid_end, 60)
                 grid(sm2.model, grid_start, grid_end, 60)) > 0.9
```

The function $\texttt{grid}(f, l, u, s)$ creates a discrete model f_d that is defined only on the grid defined by the start l, the end u and the step s. For every point $t_i = l + is, t_i \leq u$ it is $f_d(t_i) = f(t_i)$. □

While the grid materialization provides a baseline solution, it has two shortcomings. First, the analyst must guess an appropriate grid granularity. A too fine grid will produce accurate results but it will also lead to unnecessarily large discrete models and slow query processing. On the other extreme, a too coarse grid will produce inaccurate results. A better approach is to allow Plato to automatically devise the appropriate grid granularity, based on smoothness of the involved functions. [[[to Yoav and Yannis K (Priority 2): Is selection of the appropriate grid granularity a solved problem in signal processing? It sounds very close to the sampling problem: How many samples are enough?]]] <==

Still, even with automatic inference of the appropriate grid, the grid-based query processing misses the large opportunity discussed next.

Processing queries directly on model representations Statistical functions can be performed directly on the model representations, therefore reaping the benefits of the compression. In the running Example 4.1,

the correlation function query of can be executed faster directly on the frequency representation. [[[to Yannis P: Complete formula that computes correlation from frequencies.]]] <==

For each model class, Plato will internally have a corresponding implementation of each statistical function. Consequently, query processing will be automatically using the appropriate implementation (e.g., correlation on the frequency domain). The grid technique will be the last resort, in the case where Plato does not have an implementation that works directly on the compressed model representations.

[[[to Yoav (Priority 1): What do we do if we want to correlate two models that are based on different representations? Eg, consider that x is stored in a frequency representation and y is stored in an ARMA. What is the fastest algorithm for computing the correlation of x and y? There should be correlation algorithms that are better than decompressing x and y in the time domain. Are there such algorithms? If no, we should add in proposal. If yes, is it known how many options are available and what is the best option? If no, we should add in the proposal.]]] <== **Yoav:** I believe you correlate *signals*, not models. You can check whether two models are *consistent* with each other, or you can compute the *likelihood* that a model assigns to an observed signal.

As for correlating signals, your example is correct. I don't know of methods for efficiently computing the correlation between two signals that are represented in a way other than the FFT.

4.1 Any-time query processing

The analyst may elect to materialize a model representation in a way that facilitates any-time query processing and the quick evaluation of conditions. For example, consider a frequency-based representation. It can internally be represented by a sequence of frequency-amplitude pairs, sorted by the amplitude of each frequency. Consequently, given the query of Example 4.1, the frequency-based correlation algorithm may need only the first high amplitude frequencies in order to show that the correlation is above 0.9.

Generally, a model representation can be stored in sequences of data where prefixes of increasing size correspond to models of increasing accuracy. Plato will allow any-time queries that utilize the above representation property to compute results in increasing accuracy.

[[[to Yannis P: spell out syntax of anytime queries. Look if BlinkDb has the concept also.]]] <==

[[[Question to Yoav (Priority 1): For the typical model classes (ARMA, Fourier, SVD etc) is there a single and obvious ordering (eg, amplitude in Fourier) that accomodates all typical statistical functions? Or we need to state a problem: Discover the appropriate orders for various functions.]]] <== **Yoav:** I am not sure what you mean by "ordering" either in general or in the context of the Fourier transform. Do you mean something like the ordering of eigen-vectors in SVD by decreasing eigen-values?

4.2 Rewriting

[[[to all: See first reminder of this section. We can probably capture it as a rewriting case, where the way that the query runs is not by trying all pairs.]]] <==

5 Yoav's sections

6 Motivation (Yoav)

The analysis of sensor data has traditionally been the domain of signal processing or specialized areas such as image and video processing, audio processing, MRI, ultrasound, Seismology and the like. In general, most of the focus is on the real-time analysis and compression and reconstruction of the signals.

These approaches, while very successful in specialized domain, do not scale well to sensor networks that contain a large number of sensors of diverse types that collect data over a period of years. In order to make it possible to access such data collections efficiently we propose to use abstractions and techniques from the field of data-bases.

However, existing database systems lack a critical abstraction which is the *Physical Model*. In general, the data contained in a database have a one-to-one mappring to real world objects and events. for example a person might have an address, a name and a telephone number, all of which are unique entities in the world. On the other hand, the meaningful events in a video recoding from a webcam positioned over a highway are *not* the pixel values. The meaningful events might be the number of vehicles in view, the speed of vehicles in each lane or license plate numbers of the vehicle. The entities in the real world do no relate to any individual pixel or even group of pixels. They relate to *patterns* that appear across different pixels at

different times. It is by finding these patterns that we can extract useful information from the video.

7 DataBase Design principles

Database Normalization defines a set of operations for simplifying a database schema and reducing the physical size of the database. From the standpoint of information theory, one can view the act of Normalization as a type of data compression. However, while database normalization requires strict logical implication between attributes, much weaker dependencies between attributes suffices for compression.
Suppose we have a relation with four attributes A, B, C, D. In the un-normalized form we store the relation in a single table with four columns. If there are mappings $B \rightarrow A$ and $B \rightarrow (C, D)$ then we can use B as a key and break the table into two smaller tables: one for A, B and one for B, C, D.
To view the original table as a probability distribution, consider the A, B, C, D to be random variables with respect to some distribution over the rows of the table (the uniform distribution is most commonly used). We can now consider statistical dependenies between attributes. For example the implication conditions described above imply the following statistical conditions

$$[B = b \rightarrow A = a] \Rightarrow [P(A = a|B = b) = 1] \text{ and } [B = b \rightarrow (C, D) = (c, d)] \Rightarrow [P(C = c, D = d|B = b) = 1]$$

This gives a sufficient condition for compressibility, but it is far from necessary. The value of the common variable B does not have to *imply* the value of the other variables in order for compression to be possible. It is enough that there is a statistical dependence between the variables. In other words, that

$$P(A = a|B = b) \neq P(A = a) \text{ and } P(C = c, D = d|B = b) \neq P(C = c, D = d)$$

The ultimate measure of the compressibility of a relationship is the *entropy* of the relation:

$$H(P(A, B, C, D)) = -\sum_{(a,b,c,d)} P(A = a, B = b, C = c, D = d) \log_2 P(A = a, B = b, C = c, D = d)$$

According to Shannon's source coding theorem ([]) if a table is known to have this distribution over it's rows, then it requires $H(P(A, B, C, D))$ bits of storage per row.
Lossy compression TBD
In order to make use of this compression scheme we need to also store the *model* which is expressed here as P. Typically, the models will not be expressed as a big table with the probability of each possible combination of attributes. Instead, a parametric representation of the distribution as a function is typically used. Such a representation, if it fits the data well, can greatly reduce the storage and computation resources needed.
To be continuted, but give me some feedback!

8 Incremental update of the model

In a typical application we need to learn the model from the sensor data that is accumulated over time. For the solution to be practical we need to be able to incrementally update the model without re-processing all of the historical data.
We consider two scenarios:

- **Fixed distribution** This is the classical situation studied in statistics and machine learning. For some model families (specifically the exponential families) there exists compact representations of history called *sufficient statistics*. Sufficient statistics, such as the empirical mean and the empirical standard deviation, hold all of the information needed to evaluate a distribution model wrt the history.
  At the opposite extreme, with complete generality but potentially prohibitive computational demands, sits Bayesian statistics [] and online learning [**?**]. Under this approch, we keep score of the cumulative loss of each model. This approach provides universal guarantees on the regret - the cumulative loss of the method is never much worse than the cumulative loss of the best model in hind-sight.
  There is a lot of recent research on methods that lie between these two extremes - using small summaries of history to achieve performance that is close to that of the Bayesian methods.
- **Time-Varying distribution**

=======

9 Model Database Design Principles and Tools

A data warehouse designer uses well understood techniques to choose which added-value tables to materialize in the warehouse.[5] In a sense, Plato is also a data warehouse, where models provide added value on the measurements data, by revealing the real world reality behind the measurements. Similarly, a Plato warehouse designer will need to make a good choice of tables and models, based on appropriate criteria, whose application can be assisted by tools.

The first one, called *precomputation* design criterion, is similar to conventional warehouse design. It calls for computing in advance the models that are used in recurrent queries, instead of computing the models from the measurements online. The latter approach would be too slow.

The second one, which we call *consolidation*, is novel and enables higher compression of the representations. In a sense, it is a probabilistic extension of database normalization, dictating that if some models are heavily correlated, the administrator should consider consolidating them into a single model, even if this single model is not being used directly by a query. [[[I would like a better name for "consolidation"]]] <==

EXAMPLE 9.1 For instance, Example 2.3 models the sensor data by a table `sensor_models(sensor, temp_model)`, i.e., each sensor has a corresponding temporal model. Example **??** models the sensor data by a single model `full_model` that predicts the temperature based on `x`, `y` and `t`. Next, let us assume that the application often issues queries that ask for the predicted temperatures at locations other than the locations of the sensors. While it is possible for the query to compute the required predictions online from the models of `sensor_models(sensor, temp_model)`, such approach would probably be very slow. The precomputation design criterion says that, given such query workload, the model administrator must choose the single model of Example **??**.

Then let us consider an alternate scenario where the application does *not* issue queries that ask for predicted temperatures at arbitrary locations. Rather, it cares exclusively about the temperatures at the sensor sites. In such case the precomputation criterion points towards the "many models" design of Example 2.3. Yet the consolidation criterion may still dictate that it is best to choose the single model of Example **??**. The intuition is that the temperature sensors may be correlated since physics says that temperatures are correlated in sufficiently short distances. The consolidation criterion can be formally judged by comparing the entropy of the models of the "many models" design against the entropy of the "single model' design. As described below. □

[[[Database normalization was misrepresented. It is more nuanced than the simplification of the example. I now question the usefulness of fully drawing out in the example the parallel with normalization (it will waste lines to make the solid exposition). I can still do it in abstract text. However, I suggest that our entropy presentation follows on the "many models" Vs "single model" pattern set by the example. Do not worry about the temperature part. We will change it to some pollutant or something sexier.]]] <==

Suppose we have a relation with four attributes A, B, C, D. In the un-normalized form we store the relation in a single table with four columns. If there are mappings $B \to A$ and $B \to (C, D)$ then we can use B as a key and break the table into two smaller tables: one for A, B and one for B, C, D.

To view the original table as a probability distribution, consider the A, B, C, D to be random variables with respect to some distribution over the rows of the table (the uniform distribution is most commonly used). We can now consider statistical dependenies between attributes. For example the implication conditions described above imply the following statistical conditions

$$[B = b \to A = a] \Rightarrow [P(A = a|B = b) = 1] \text{ and } [B = b \to (C, D) = (c, d)] \Rightarrow [P(C = c, D = d|B = b) = 1]$$

This gives a sufficient condition for compressibility, but it is far from necessary. The value of the common variable B does not have to *imply* the value of the other variables in order for compression to be possible. It is enough that there is a statistical dependence between the variables. In other words, that

$$P(A = a|B = b) \neq P(A = a) \text{ and } P(C = c, D = d|B = b) \neq P(C = c, D = d)$$

[5] Such tables are often called *materialized views*.

The ultimate measure of the compressibility of a relationship is the *entropy* of the relation:

$$H(P(A,B,C,D)) = -\sum_{(a,b,c,d)} P(A=a,B=b,C=c,D=d)\log_2 P(A=a,B=b,C=c,D=d)$$

According to Shannon's source coding theorem ([]) if a table is known to have this distribution over it's rows, then it requires $H(P(A,B,C,D))$ bits of storage per row.
Lossy compression TBD
In order to make use of this compression scheme we need to also store the *model* which is expressed here as P. Typically, the models will not be expressed as a big table with the probability of each possible combination of attributes. Instead, a parametric representation of the distribution as a function is typically used. Such a representation, if it fits the data well, can greatly reduce the storage and computation resources needed.
To be continuted, but give me some feedback!

10 Incremental update of the model

In a typical application we need to learn the model from the sensor data that is accumulated over time. For the solution to be practical we need to be able to incrementally update the model without re-processing all of the historical data.
We consider two scenarios:

- **Fixed distribution** This is the classical situation studied in statistics and machine learning. For some model families (specifically the exponential families) there exists compact representations of history called *sufficient statistics*. Sufficient statistics, such as the empirical mean and the empirical standard deviation, hold all of the information needed to evaluate a distribution model wrt the history.
  At the opposite extreme, with complete generality but potentially prohibitive computational demands, sits Bayesian statistics [] and online learning [**?**]. Under this approch, we keep score of the cumulative loss of each model. This approach provides universal guarantees on the regret - the cumulative loss of the method is never much worse than the cumulative loss of the best model in hind-sight.
  There is a lot of recent research on methods that lie between these two extremes - using small summaries of history to achieve performance that is close to that of the Bayesian methods.
- **Time-Varying distribution**