Online Filtering, Smoothing and Probabilistic Modeling of Streaming data

Bhargav Kanagal
bhargav@cs.umd.edu
University of Maryland

Amol Deshpande
amol@cs.umd.edu
University of Maryland

Abstract

In this paper, we address the problem of extending a relational database system to facilitate efficient real-time application of dynamic probabilistic models to streaming data. We use the recently proposed abstraction of *model-based views* for this purpose, by allowing users to declaratively specify the model to be applied, and by presenting the output of the models to the user as a *probabilistic database view*. We support declarative querying over such views using an extended version of SQL that allows for querying probabilistic data. Underneath we use *particle filters*, a class of sequential *Monte Carlo algorithms* commonly used to implement dynamic probabilistic models, to represent the present and historical states of the model as sets of weighted samples (*particles*) that are kept up-to-date as new readings arrive. We develop novel techniques to convert the queries on the model-based view directly into queries over *particle tables*, enabling highly efficient query processing. Finally, we present experimental evaluation of our prototype implementation over sensor data from the Intel Lab dataset that demonstrates the *feasibility* of online modeling of streaming data using our system and establishes the advantages of such tight integration between dynamic probabilistic models and database systems.

1 Introduction

Enormous amounts of streaming data are being generated everyday by measurement infrastructures that continuously monitor a variety of things from environmental properties using sensor networks [29] to behavior of large computational clusters [19]. To fully harvest the benefits of this extensive monitoring, we must be able to process and analyze such data streams in real-time. In recent years, there has been much work on *data stream management systems* [6, 7, 5, 31] that can process high-rate data streams in real-time and continuously evaluate SQL queries over them. A large class of commonly used stream processing tasks, however, cannot be expressed as SQL queries and thus cannot benefit from these advances in stream data processing. Examples of such tasks include:

- **Inferring *hidden* variables:** In several real-world data streams, the attributes of interest may not be directly observable (e.g. *working status* of a remotely located wireless sensor), or it may be very expensive to measure them (*light* on a Berkeley Mote [12]). A common processing task over such data streams is to continuously *infer* the value of the hidden variables using the observed data. Hidden Markov models [36], or variations thereof, are often used for this purpose. These types of models allow us to combine prior domain knowledge about the system behavior with the actual observations to compute the most likely values of the hidden variables (Section 2.1).

- **Eliminating measurement noise:** Data Streams generated by distributed measurement infrastructures like sensor networks or GPS devices, are invariably noisy; this could be because of calibration effects (e.g., temperature sensors typically report *voltages* that are then converted into Celsius), errors due to poor coupling or analog-to-digital conversion, inaccuracies due to non-robust measurement techniques that fail in harsh environments (e.g., multi-path propagation errors that occur in GPS in urban settings), or inherent flaws with mass-produced sensing devices. Removing measurement noise is perhaps the most important first step when analyzing such data streams or processing user queries over them. Analytical filtering techniques such as Kalman filters [43] (Section 2.2), and its extensions like extended Kalman filters and unscented Kalman filters, are commonly used for this purpose in a wide variety of domains.
- **Probabilistically modeling high-level events from low-level sensor readings:** Automatically recognizing higher level events such as user activities through use of unobtrusive sensing technologies is considered a key in the field of *Ubiquitous computing* [9, 34, 26]. For instance, Patterson et al. [34] demonstrate how the *transportation mode* of a user can be learned using GPS readings, which they then use to design a guiding device for cognitively impaired people. Increasing deployments of large-scale sensing infrastructures will enable many such applications in the near future. Ideally, we would like to perform this type of modeling in real-time as the data is being generated and streamed into the system; the application developers can then be provided access to these inferred events (subject to privacy policies) directly in a streaming fashion, so they can provide user services.

There are several other common stream processing tasks such as predictive modeling and extrapolation to fill up missing values, identifying temporal or spatial trends and patterns in the data, novelty and anomaly detection and so on, that cannot be expressed as SQL queries. In most such applications, the majority of the analysis and querying is typically done outside the database system, leading to much repetition of functionality and highly inefficient execution. We start by presenting a motivating application in detail in the following subsection.

1.1 Motivating Application

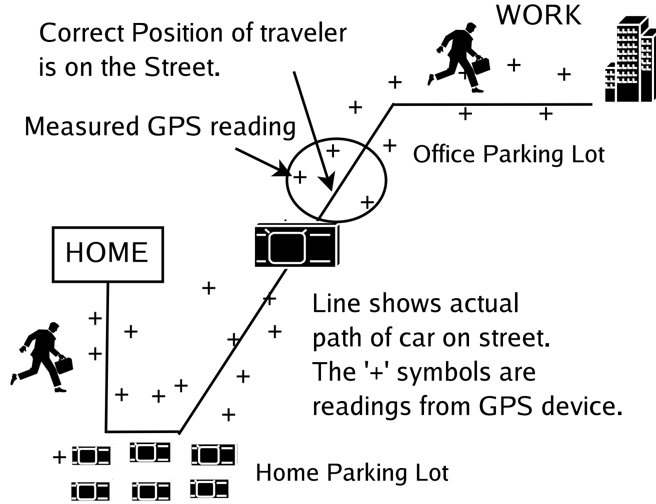
We consider the following stream processing task of inferring *transportation mode* and velocity of an urban traveler from GPS data which was presented in [34]. GPS has become a vital global utility, indispensable for modern navigation. Patterson et al in [34] show that inferring these transportation modes can help in building rich predictive models of human behavior which is a central theme in ubiquitous computing. Figure 1(a) shows a simulated GPS trace of a person going from home to work and Figure 1(b) plots the trace on the road map. Note that the GPS data contains inherent measurement noise (due to loss of signal, triangulation error, multi-path propogation), prominent in urban settings. For instance, we can locate the user at the parking lot but the GPS readings indicate the user is elsewhere. Any query executed on this table is bound to have errors as these readings represent the "raw" measurement values and *not* the underlying location of the person that is of interest. Hence, we cannot the use the table of GPS readings directly for querying purposes.

As Patterson et al show in [34] , we can use a dynamic probabilistic model to both clean the data and to infer the velocity and the transportation mode from this GPS data. We will discuss the specific form such a model might take in Section 2.3, but in essence, we can model the state of the system at time, t, using three variables, $(mode_t, loc_t, velocity_t)$ and obtain a *probability distribution* over the possible states using the DPM.

Figure 1(iii) shows a possible output of the model presented to the user as a probabilistic view. This view provides a consistent picture of the underlying model to the user with all the variables of interest. Using this system, the user is free to ask rich SQL queries over the output of the models. It must be noted here that

Time	Position	
	x	y
...	...	...
22	32.89	34.40
23	32.93	32.23
24	-	-
25	57.07	48.01
26	65.49	53.86
27	-	-
28	-	-
29	106.4	80.0
30	107.12	80.51
31	107.12	80.51
...	...	...

(a) Input GPS Data



(b) Plot of GPS trace

Time	Position		*Velocity*		*Mode*
	x	y	v_x	v_y	
...	...	...	...	...	...
22	32.50	35.45	0.15	2.50	Walking
23	32.50	32.20	0.08	2.45	Walking
24	44.50	40.5	12.5	8.3	Driving
25	56.56	48.25	11.0	8.2	Driving
26	68.54	56.5	12.3	8.1	Driving
27	80.32	64.5	12.5	7.9	Driving
28	92.44	72.5	12.1	8.5	Driving
29	104.4	80.5	12.5	8.1	Driving
30	106.50	80.5	2.1	0.22	Walking
31	108.8	80.8	2.1	0.3	Walking
...	...	...	...	...	...

(c) Output Probabilistic View (Our System)

Figure 1: (a,b) GPS trace of the urban traveler traveling from home to work. (c) The output of a dynamic probabilistic model (used to both clean the GPS data and to infer the velocity and transportation mode) can be presented as a user-queriable deterministic database table.

queries can also be posed on $velocity$ and the $mode$, which were the hidden attributes inferred using the DBN.

In this paper, we present an extensible system that exploits the commonalities between many of these tasks to natively support them inside a relational database system. Most of the aforementioned tasks can be seen as applications of specific instances of *dynamic probabilistic models* (DPMs) to streaming data [33,

30][1]. We use the recently proposed abstraction of *model-based views* [13] to push the application of a wide range of DPMs to streaming data inside a relational DBMS, thus enabling easy application of these tasks.

1.2 Contributions

The salient contributions of our work are as follows:

- We extend the abstraction of model-based views [13] to present the output of a DPM to the users as *probabilistic tables*, leading to intuitive user interaction with the system.
- We exploit the structure of *particle filters* (a widely-applicable *sequential Monte Carlo inference* technique) to efficiently store the probabilistic output of a DPM as a set of weighted samples (called *particles*). Our internal representation also naturally captures many of the attributes correlations present in the data.
- We design novel techniques to convert queries posed over a single DPM-based view (including aggregate queries) into queries over our internal particle-based representation, allowing us to exploit the existing query processing machinery.
- Our experimental results show that we can achieve sufficient accuracy with a few particles (about 100) and that the time taken to process large number of particles (about 1000) is quite reasonable. This makes our system feasible for online modeling of streaming data.

The rest of the paper is organized as follows. We begin with an overview of dynamic probabilistic models in Section 2. We describe the abstraction of DPM-based views in Section 3 and present a detailed description of our system and the algorithms used in Sections 4-8. We conclude with an experimental evaluation of our prototype implementation in Section 9.

2 Dynamic Probabilistic Models

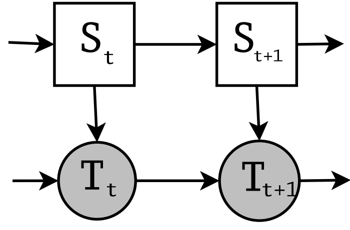
Dynamic probabilistic models are widely used in practice to model complex real-world stochastic processes and to reason about them. They form an active area of research in the machine learning community [22, 16, 33, 18, 30] and are playing an increasingly important role in the design of machine learning algorithms. The simplest and most widely used examples of dynamic probabilistic models are *hidden Markov models (HMMs)* and *linear dynamical systems* (better known as Kalman filter models (KFMs)). We illustrate dynamic probabilistic models by describing some of the simple applications of HMMs and KFMs and then describe a generalized graphical representation for DPMs.

2.1 Hidden Markov Models

Hidden Markov models (HMMs) are used in a variety of applications like speech recognition [36], bioinformatics [15] and fault detection [23, 46]. In short, HMMs are used to infer the values of unobservable (hidden) state variables from the imprecise observations that are made about *related* variables.

We will illustrate HMMs using a simple fault detection application. Fault detection in mass produced, low cost sensor nodes is an important research problem and is widely studied in literature [23, 38, 46]. Let us consider an example of a single sensor, possibly faulty, that is measuring temperatures in a room and transmitting them to a central database server. We want to know whether the sensor is working correctly

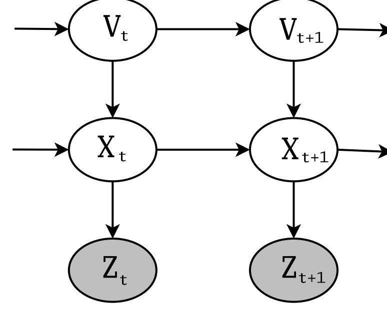
[1] In this paper, we assume that DPMs are equivalent to the class of *dynamic Bayesian networks*. In some literature, DPMs are considered to be a more general class of models than the ones we consider here.



$$p(T_t|T_{t-1}, S_t) = \begin{cases} N(T_{t-1}, \sigma) & S_t = Wo \\ U(min, max) & S_t = Fa \end{cases}$$

$$p(S_{t+1}|S_t) = \begin{array}{c|cc} & Wo & Fa \\ \hline Wo & 0.99 & 0.01 \\ Fa & 0.01 & 0.99 \end{array}$$

(i) AR-HMM [2]



$$p(V_{t+1}|V_t) = N(V_t, \sigma_V)$$
$$p(X_{t+1}|X_t, V_{t+1}) = N(X_t + V_{t+1}, \sigma_X)$$
$$p(Z_{t+1}|X_{t+1}) = N(X_{t+1}, \sigma_Y)$$
$$Priors: \ p(V_0) \ and \ p(X_0)$$

(ii) KFM

Figure 2: Graphical representations of DPMs. (i) Using an HMM for fault detection; (ii) Using a KFM for velocity and location estimation.

(so that we could ignore the erroneous readings produced by faulty sensors). The only information we have about the sensor are the temperature readings it transmits. We can use an HMM to infer the sensor's status from these readings alone. In this example, the *hidden variable* (that we cannot measure) is the status of the sensor, which can be in two states, $Working$ (Wo) or $Failed$ (Fa). The *observed readings* are the temperatures measured by the device.

Figure 2(i) shows the graphical representation of the HMM we can use to model the state of such a system. The shaded node denotes the observed temperature at time t (denoted by T_t), whereas the unshaded node denotes the hidden *status* variable that captures whether the sensor is working or not (denoted by S_t). The prior knowledge about the behavior of the system (that is used to determine whether the sensor has failed) can be captured by three *conditional* probability distributions:

- $P(T_{t+1}|T_t, S_{t+1} = Working)$**:** This distribution captures the expected behavior of the sensor when the sensor is functioning correctly. For instance, from prior knowledge about the process, we expect that at time $t + 1$, the sensor should return values around T_t plus a small (Gaussian) noise. This distribution can also be learned from historical data.
- $P(T_{t+1}|T_t, S_{t+1} = Faulty)$**:** This distribution captures the expected behavior when the sensor is faulty. One simple way to model something like this would be to assume that the sensor might return any value from 0 to 100 uniformly independently of the actual temperature. Clearly the faulty behavior depends on the nature of the sensor.
- $P(S_{t+1}|S_t)$**:** Figure 2(i) shows a possible table for this that captures the prior knowledge that the sensor has a small probability of failing. The table indicates that if the sensor was working at time t, then the probability that it will fail in the next time instant is 0.01 and if the sensor has failed now, the probability that it will work the next time instant is 0.01. Once again, the actual probabilities depend on the nature of

[2] AR-HMMs (Auto-Regressive HMMs) are a specific class of HMMs.

the sensor and possibly the manufacturing process; for most devices, this type of information is typically available.

These conditional distributions form the *parameters* of the HMM. By combining them with the observed temperatures from the data stream, HMMs provide mechanisms to infer the best possible estimate of the hidden variables (in our case, the status of the sensor at various times) using the observed measurements. Some of the common *analytical* algorithms used for this purpose are *forward-backward* and *Vitterbi* algorithms [36]. These algorithms can be seen as special cases of general *inference* algorithms designed for dynamic probabilistic models [40]; we describe one such inference technique in Section 6.1.

We would like to note that this model is not appropriate for predicting future temperature values from current temperature and is appropriate only for fault detection. A model that can be used to predict future temperatures is shown in Figure 3(i).

2.2 Linear Dynamical Systems

A *linear dynamical system*, more commonly known as the *Kalman filter model (KFM)*, is another widely used dynamic probabilistic model. We illustrate KFMs using the following application. We are interested in computing the position and velocity of a car based on continuous observations of the position of the car made by an inaccurate GPS device.

Here, velocity is a hidden variable that is not being measured directly. Furthermore, the actual position is also not known because of the inherent measurement noise in GPS data. In this case the state of the car at time t is denoted by $[x_t, v_t]$, where x_t denotes the *true* location of the car (assuming one-dimensional motion) and v_t denotes the velocity. Let z_t denote the *observed* location of the car.

Figure 2(ii) shows a pictorial representation of the KFM that can be used in this application. (This model was described by Murphy [33]). Similar to the earlier example, we can summarize our prior knowledge about the process using the following equations; these equations can be easily recast as conditional probability distributions (shown in Figure 2(ii)).

$$\begin{aligned} z_t &= x_t + \mathcal{N}(0, W_1) \\ x_{t+1} &= x_t + v_{t+1} + \mathcal{N}(0, W_2) \\ v_{t+1} &= v_t + \mathcal{N}(0, W_3) \end{aligned}$$

The first equation specifies how the measurement noise (which is assumed to be a zero-mean Gaussian with covariance W_1) affects the observed locations, whereas the latter two equations encode the movement of the object and the random perturbations that the location and velocity might be subject to.

Kalman filter actually refers to a specific analytical *inference algorithm* for the LDS model [43]. Given the observed variables and the conditional distributions, this algorithm can be used to obtain a distribution over the hidden variables (*velocity* and *true location* in our case) and once again, it can be seen as a special case of general inference algorithms for DPMs [25]. We will continue to call this model the *Kalman filter model* (KFM).

2.3 DPMs & Graphical representation

Generally speaking, a dynamic probabilistic model (DPM) is used to compactly represent the stochastic evolution of a set of random variables over time [16]. DPMs are typically represented using a directed graphical structure as shown in Figure 3(i), where the graph structure captures the complex interdependencies between the variables of the process. We will illustrate the graphical representation using the model shown in Figure

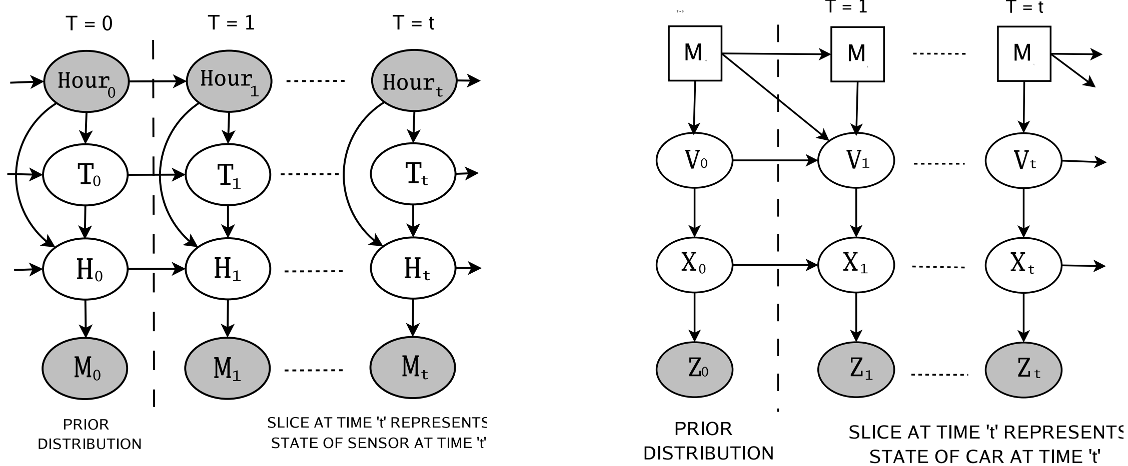


Figure 3: (i) Graphical representation of the BBQ DPM used for modeling Intel Lab data (Section 9); (ii) A generic DPM used for Transporation Mode Estimation application discussed in Section 1.

3(i), which we call "BBQ DPM" [12]. This model is used to infer hidden temperature values using observed humidity values in a sensor network and also remove noise from the observed humidity values (Section 9). Figure 3(ii) is another generic DPM that can be used for the transportation mode estimation example discussed in Section 1. The details of the graphical representation are illustrated below.

Nodes: The nodes of the graph represent the attributes of the system being modeled (that are modeled as random variables). In Figure 3(i), the attributes of the system being modeled are temperature T_t and humidity H_t. The measured humidity M_t and hour of the day $hour_t$ are observed quantities. By convention, the observed nodes are shaded while the hidden nodes are clear.

Edges: The directed edges represent "causality". In Figure 3(i), the temperature at time t influences the temperature at time $t + 1$. Similarly the temperature at time t influences the humidity at time t. The degree of causality is indicated by the *conditional probability distribution function* (CPD). The CPD of node X is indicated by $P(X|Pa(X))$ where $Pa(X)$ denotes the parents of node X.

Time: Time is represented in a DPM through use of *vertical slices*. Each vertical slice of the graphical model corresponds to the state of the system at a given time instant. As time advances, we can *unroll* the model by repeating the structure and parameters of the model as shown in Figures 3 (i),(ii).

CPD: The key parameters of a DPM are the conditional probability distributions that encode our knowledge about the system being monitored, and its evolution over time. We need three sets of probability distributions to fully specify a DPM:

- The prior (unconditional) probability distributions over the variables in the first slice (that may not have any parents).
- The conditional probability distributions that encode the knowledge about how the state at time $t + 1$ depends on the state at time t.
- The conditional probability distributions that encode the knowledge about the how the observation at time t depends on the state at time t.

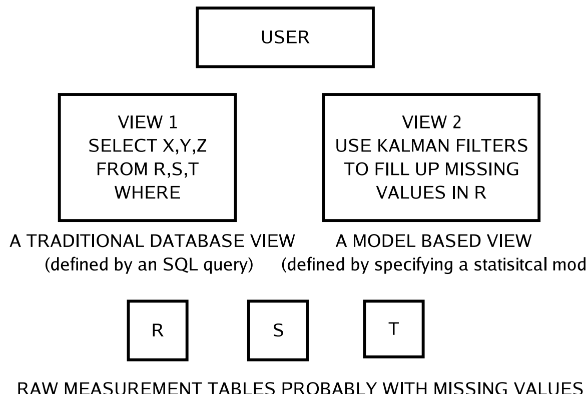


Figure 4: Abstraction of model-based views

Typically it is assumed that the variables at time t depend directly only on the variables at time t and $t-1$ (*Markov* assumption), and hence a *2-slice* representation (as shown in Figures 2(i) and (ii)) is usually sufficient. In Figure 3(i), the CPD of node T_{t+1} depends on T_t and the hour of the day h_{t+1}. This is because hourly change in temperature depends on the hour of the day. Temperature increases during the day, decreases at night by a magnitude that also depends on the hour. The CPD of the humidity node H_t is similar.

2.3.1 Inference

The ultimate goal of modeling a stochastic process using a DPM is to obtain a *posterior* distribution over the hidden variables of the graphical model, given the observed measurements. This task is called *inference*. Several inference algorithms have been developed for efficient inference in special cases (e.g. Kalman Filters), and many general purpose inference techniques (e.g. *junction tree algorithm*) are also known. We present one such general purpose algorithm, based on Monte Carlo techniques, in Section 6.1. The output of the inference algorithm is typically a probability distribution over the hidden variables (although some inference algorithms may only provide expected or most likely values of the hidden variables).

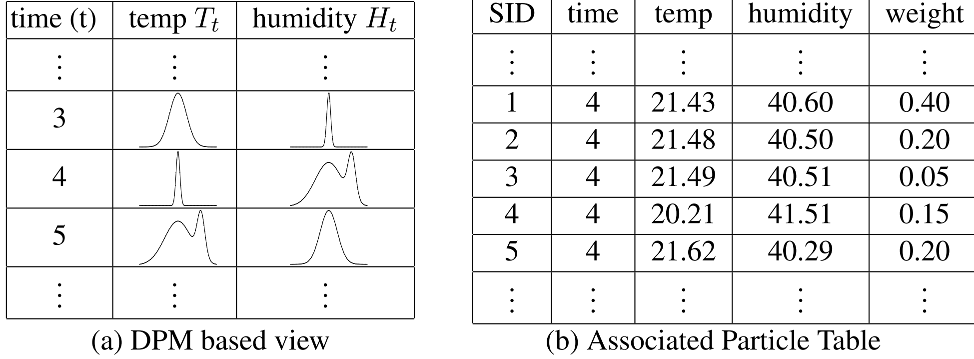
2.3.2 Learning

The CPD parameters of the DPM are typically not known to the user and hence need to be learned from training data. Maximum Likelihood Estimation (MLE) is one of the popular statistical methods used for learning parameters of the CPDs. Given learning data, MLE estimates the values of the parameter values, θ that maximizes the likelihood of observing the learning data. In other words, we determine the parameter θ that makes the learning data “most likely”. Details of MLE are described in Section 7.

In the next section, we present our proposal for presenting the output of a DPM to the users.

3 DPMs as Database Views

We use the abstraction of *model-based views*, proposed in our prior work [13], to present the output of a DPM to the users. This abstraction is analogous to the well-known abstraction of *database views* which allows creating *virtual tables* using an SQL query over one or more database tables. *Model-based views*



time (t)	temp T_t	humidity H_t
⋮	⋮	⋮
3		
4		
5		
⋮	⋮	⋮

(a) DPM based view

SID	time	temp	humidity	weight
⋮	⋮	⋮	⋮	⋮
1	4	21.43	40.60	0.40
2	4	21.48	40.50	0.20
3	4	21.49	40.51	0.05
4	4	20.21	41.51	0.15
5	4	21.62	40.29	0.20
⋮	⋮	⋮	⋮	⋮

(b) Associated Particle Table

Figure 5: (a) DPM-based views contain probabilistic attributes; (b) Particle-based representation of the view (only particles corresponding to the second tuple, *time* = 4, are shown for clarity)

further this abstraction and allow creating such virtual tables using a statistical model instead (Figure 4). Several examples of model-based views based on non-parametric statistical models like *linear regression* and *interpolation* are described in [13].

We extend this abstraction by allowing views to be defined using DPMs instead. Figure 5(a) shows the schema of the view that would be presented to the user with the BBQ DPM model (Section 2.3) that infers temperature from observed humidity values. As we can see, the schema contains all the state variables in the DPM as attributes along with a *time* attribute. As with traditional database views, this is a virtual table and it may or may not be *materialized*.

Note that the nature of DPMs forces these to be *probabilistic* views in that the attributes of this virtual table may be probabilistic. For instance, in Figure 5(a), the *temperature* attribute is not known with certainty since the DPM only provides a probability distribution over it. Although the above DPM based view shows only continuous variables, DPM views can also have discrete variables. (e.g. *status* attribute in HMM based view presented to the user in the Fault detection example (Section 2.1)).

The issue of querying and representing such probabilistic data has received much attention in recent years [4, 24, 8, 44, 11, 2, 39], and some of the challenges we face form active research focuses in that area. We plan to utilize the techniques developed in that work to a large extent in building our system, especially language extensions for querying such tables. We currently allow querying single table DPM-based views using an extended version of SQL with the following features:

- $\mu(X)$: We allow the users to specify operations on the expected values of probabilistic attributes using this extensions. Thus, a predicate such as $\mu(temp) > 30$ indicates that the condition is on the mean value of the temperature attribute.
- $with\ confidence\ c$: This construct allows the users to specify a minimum confidence in the result tuples returned to the users.
- We allow queries with aggregates such as AVG, MIN, MAX and NN (Nearest Neighbor).
- Although the query processing itself is cognizant of the probability distributions and the correlations, if the attributes of the final result are probabilistic, the user is presented with either the expected values (μ) or the most likely values (in case of discrete attributes) instead.

One significant way DPM-based views differ from prior work in this area is that DPM-based views exhibit complex and strong attribute correlations that can not be ignored during query processing. Most of the probabilistic databases either assumes independence or severely restrict the correlations that can be represented.

We differentiate between two types of correlations:

- **intra-tuple correlations:** that exist between attributes of a single tuple. For example, in Figure 5(a), *humidity* and the *temperature* attributes are correlated.
- **inter-tuple correlations:** that exist between attributes of different tuples. For example, *temperatures* at times t and $t+1$ are likely to be highly correlated with each other.

Our internal representation (that we discuss next) currently captures the intra-tuple correlations, and the query results are also affected by it. Inter-tuple correlations, on the other hand, are harder to capture and we currently ignore those during query processing; in future, we plan to develop intuitive ways of representing and querying such correlations.

3.1 Particle-based representation

We use a representation based on *weighted samples* (called *particles*) to store the DPM-based views internally. This not only allows us to handle the complex, continuous, and correlated probability distributions that may be generated during probabilistic modeling, but also forms the basis for our inference technique.

Definition: *A* particle *is a weighted sample drawn from a probability distribution. The weight associated with the sample represents its likelihood of occurence in the distribution.*

To represent a DPM-based view as a relational table with deterministic attributes, we essentially maintain a set of particles for each tuple in the view. These particles are drawn from the joint distribution over all the attributes in the view. Figure 5(b) shows an example set of particles corresponding to one of the tuples in the view. Clearly the accuracy of such a representation depends on the number of particles used (N, a system parameter). It can be shown theoretically that the accuracy of the representation is proportional to $1/N$ [14]. The schema of this table, called *particle table*, consists of the attributes of the view along with a SampleID attribute (*SID*), and a *weight* attribute.

Given such a particle table, the expected (or most likely) values of the attributes are computed by taking weighted averages over the particles. For example, the expected value of the $temperature$ attribute at time 4 is given by (Figure 5(b)):

$$T_4 = \sum_{i=1}^{N} (T_4^i * w_4^i) = 21.43 * 0.4 + \cdots + 21.62 * 0.2 = 21.28$$

Similarly, we can compute expected value of humidity attribute. Since the particles are drawn from the joint distribution, the intra-tuple correlations are naturally captured in this representation.

Populating the Particle Tables

At the start of the process (at $time = 0$), the particle table is initially populated by sampling over the prior distributions of the attributes. After that, the particles are directly generated, and in some cases updated, by the inference algorithm that we use (Section 6).

4 System Design

The schematic diagram depicting the architecture of our system is shown in Figure 6. Suppose that the user is interested in modeling a data stream based on a suitable probabilistic model. The following sequence of steps occur, we will look at each of the operations in detail.

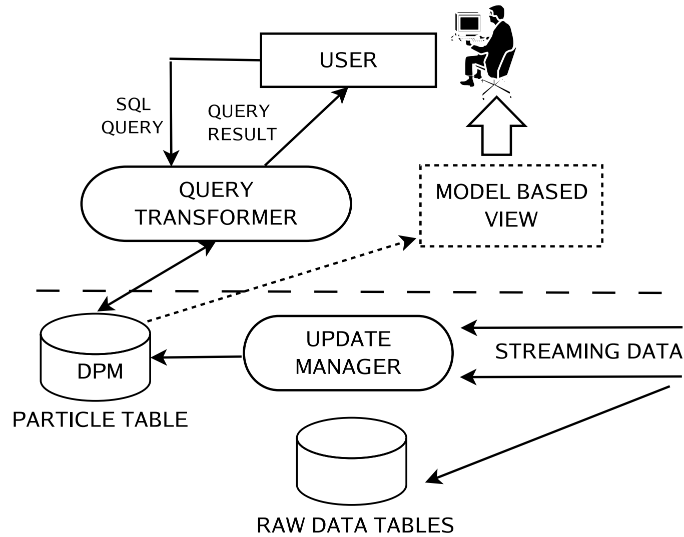


Figure 6: Architecture of our system

1. The user issues a request to the View Manager to create the DPM based view by providing a *create view definition* statement (Section 4.1). This specifies the DPM to be used, data stream to be modeled, training data to learn the DPM.
2. If the parameters are to be learned using training data, the *learning module* is invoked over the training data (Section 7).
3. A *particle table* is created and populated using the prior distributions specified along with the view definition (Section 3.1).
4. The user interacts with the DPM based view by issuing SQL-style queries (extended to deal with probabilistic data). A *query transformer* intercepts these queries and converts them into SQL queries over the particle table that are then executed using the traditional query processor (Section 5).
5. The particle table is continuously updated by the *Update Manager* in order to keep it consistent with the incoming data stream (Section 6.1)

4.1 Specifying DPM-based Views

To create a DPM-based view over a stream, the user is required to specify the following details:
– The *schema* of the view.
– The *data stream* to be modeled.
– The *dynamic probabilistic model (DPM)* to be used to model the data.
The generic view definition statement to create a DPM-based view is as follows.

```
CREATE VIEW <name_of_view> <Schema> AS
DPM <DPM_config_in_file>
<TRAINING_DATA <SQL_query_for_training_data>>
STREAMING DATA <SQL_query_for_streaming_data>
```

```
# Node Properties
numNodes: 4
hidden: {1,3}
discrete: {1,3}
node(1): ['Wo' 'Fa']
node(3): ['Wo' 'Fa']

# Graph adjacency matrix
graph: [0 1 1 0;
        0 0 0 1;
        0 0 0 1;
        0 0 0 0]

# CPDs of Nodes
cpd(1):[1;0];
cpd(2):(val(1),[N(50,0.05);
                U(-100,+100)]);
cpd(3):(val(1),[[0.99;0.01];
                [0.01;0.99]]);
cpd(4):(val(3),[N(val(2),0.05);
                U(-100,+100)]);
```

(i) Configuration file for the HMM model

```
# Node Properties
numNodes: 6
hidden: {1,2,4,5}
discrete: {}

# Graph adjacency matrix
graph: [0 1 0 1 0 0;
        0 0 1 0 0 0;
        0 0 0 0 0 0;
        0 0 0 0 1 0;
        0 0 0 0 0 1;
        0 0 0 0 0 0]

# CPDs of nodes
cpd(1): N(0,0.05);
cpd(2): N(0,0.05);
cpd(3): N(val(2),0.05);
cpd(4): N(val(1),0.05);
cpd(5): N(val(4)+val(2),0.05);
cpd(6): N(val(5),0.05);
```

(ii) Configuation file for the KFM model

Figure 7: (i) and (ii) Sample configuration files for the two DPM-based views shown in Figures 2 (i) and 2(ii)

Syntax	**Definition**
$val(i)$	Variable modeled by node i
$cpd(i)$	CPD of node i
$N(\mu, \sigma)$	Normally distributed CPD with mean μ and variance σ
$U(a, b)$	Uniformly distributed CPD with range $[a, b]$
$[p_1; p_2; p_3]$	Discrete distribution that has probability p_1 of being in first state, p_2 in the second state and p_3 in the third state.
$(val(i), [s_1; s_2])$	Discrete CPD with 2 possible states that takes state s_1 if $val(i)$ is in the first state and state s_2 if $val(i)$ is in the second state

Figure 8: Conventions used to specify DPM-based views

As we can see, the format for specifying views has been kept close to the view definition statement used for traditional database views. The first line of the statement specifies the schema of the view, including its name and its attributes. The fourth line specifies the data stream to be modeled (an SQL query over a stream can be used if only a portion of the data stream is of interest). The structure and the parameters of the DPM itself are specified using a *configuration file* that is provided with the view definition. Figure 7 shows two examples of such configuration files for the two models presented in Section 2. The configuration file

contains the following details about the model:

Properties of attributes in the DPM, including whether the nodes are hidden or observed, continuous or discrete, and the set of values they can take if they are discrete. Attributes corresponding to two slices of the DPM (for times 0 and 1) are typically specified.

Adjacency matrix of the graphical representation of DPM. The edges are assumed to be directed from the node corresponding to the row to the node corresponding to the column. Note that the graphical representation for a DPM is required to be acyclic.

CPDs Prior and conditional probability distributions (Section 2.3) for each of the nodes in the graph. This is perhaps the most complex part of the DPM specification. We allow the users to specify CPDs using one of two ways.

- *Using a set of pre-defined probability distributions:* Figure 8 shows the distributions we currently support. For example, $N(\mu, \sigma)$ represents a normal distribution with mean μ and standard deviation σ. $[p_1; p_2; p_3]$ represents a discrete distribution of a random variable with three states where p_i's represent the probabilities of the variable being in each of the three states. Similarly, $(val(i), [\alpha; \beta])$ represents a discrete CPD that depends on $val(i)$. If $val(i)$ is in its first state, then it follows the distribution α and if $val(i)$ is in its second state, then if follows the distribution β and so on.

  For example, $cpd(1)$ in Figure 7 (ii) specifies that the CPD of node 1 (prior distribution on the velocity variable) is normally distributed with mean 0 and a variance of 0.05. Similarly, the CPD for node 3 in Figure 7 (ii) is given by $N(val(2), 0.05)$, which indicates that node 3 is normally distributed, its mean value is the value of node 2 and its variance is 0.05. Node 3 of the HMM, in Figure 7 (i) has a discrete distribution that was specified using a transition probability matrix in Section 2 (i). We represent that distribution in the following way. If node 1 at time t was in $'Wo'$ state, then the distribution of node 3 is given by the discrete binary distribution [0.99 0.01]. Otherwise, the distribution of node 3 would be [0.01 0.99].

- *By providing a java module file that supports an appropriate API:* If the probability distribution to be specified is not among the ones supported above, then we allow the user to provide the distribution in the form of a java class file. The class must be implemented to support a pre-defined API. We will discuss the details of the API in Section 8.

Instead of specifying the parameters explicitly using the configuration file, the user may instead specify a training dataset from which to learn the parameters (*line 3* in the view creation syntax). Learning algorithms are described in Section 7.

5 Query Processing

Query processing over DPM-based views is simplified as a result of the internal particle-based representation that our system uses. A probabilistic query is executed over DPM-based views by first converting it into query over the corresponding particle tables, and then using an existing query processor (and query optimizer) to execute the new query. This approach not only minimizes the number of changes that we have to make to the underlying database system, but also results in highly efficient query execution. We first describe in detail, the query transformation process for the case of simple *select-project* queries over a single DPM-based view. We then look at some of the issues involved in transforming more complex aggregate queries (Section 5.2). The transformation process for aggregate queries especially for aggregates such

as MIN and MAX is quite complex and we will look at a few examples of the transformed queries. For simplicity of discussion, we assume throughout this section that the attributes of the view, other than the *key* attributes, are continuous. The extensions for handling discrete attributes are quite straightforward.

Algorithm 1 Converting single-table select-project queries

Require: User SQL Query (Q) on the model based view.
1: **for** attribute i in SELECT clause in Q **do**
2:     **if** i is a key attribute **then**
3:         Add i to the SELECT clause of Q'
4:     **else**
5:         Add $sum(i * weight)$ to the SELECT clause of Q'
6: Replace occurrences of *DPM-based view* in Q with *particle-table* in the FROM clause.
7: Add *key(DPM based view)* to the GROUP BY clause of Q'
8: **for** each predicate α in WHERE clause of Q **do**
9:     **if** α involves only *key* attributes **then**
10:         Add α to the WHERE clause of Q'
11: HAVING Clause of $Q' \leftarrow$ (confidenceQuery $> c\%$)

Consider a DPM-based view that infers the *temperature* measured by a set of sensors from the observed humidity values(Section 2.3). The schema of this view is given by:

bbqview(time, id, temp, humidity)

where id denotes the sensor identifier and $temp$ is the temperature of the sensor. The unique *key* for this view is *(time, id)*, and the schema of the corresponding particle table is:

particle-table(SampleID, time, id, temp, humidity, weight)

We will illustrate the query transformation process using this example DPM-based view.

5.1 Select-Project SQL queries

Figure 9 shows two examples of query transformation over this view. The first query simply asks for the temperatures measured by all sensors. As we discussed in Section 3, if the final result attributes in a query are probabilistic, the expected values are returned instead. As shown in Figure 9, the corresponding query over the particle table simply groups the particles by the key attributes and returns a weighted average of the temperature attribute.

Algorithm 2 Constructing the *confidence* query

Require: User SQL Query (Q) on the model based view.
1: Construct Query Q'' such that,
2: SELECT clause of $Q'' \leftarrow$ sum(weight)
3: FROM clause of $Q'' \leftarrow$ *particle-table p2*
4: WHERE clause of $Q'' \leftarrow$ WHERE clause of Q
5: Add ‘p1.key = p2.key’ predicates to WHERE clause of Q'

The second query specifies a predicate over a probabilistic attribute and a confidence value that specifies the minimum confidence required in the result tuples (a default confidence value is assumed if the user doesn’t specify one). In such a case, the query transformer constructs two queries, a *result* query and a

(i)

```
(O) SELECT id, time, temp FROM bbqview

(C) SELECT id, time, sum(temp*weight)
    FROM particle-table GROUP BY id, time
```

(ii)

```
(O) SELECT id, temp FROM bbqview
    WHERE temp > 20 AND time = 5
    WITH CONFIDENCE 0.95

(C) SELECT id, sum(temp*weight)
    FROM particle-table p1
    WHERE time = 5 GROUP BY id
    HAVING 0.95 <
    ( SELECT sum(weight)
      FROM particle-table p2
      WHERE p1.id = p2.id AND
      temp > 20 AND time = 5 )
```

Figure 9: Two examples of Query Transformation. (O) denotes the original query on the model-based-view, and (C) denotes the queries on the particle table.

confidence query, that are then merged into a single query as shown in the figure. The result query generates the output as desired in a fashion similar to the first query, whereas the confidence query makes sure that only tuples with sufficient confidence are returned back to the user.

Algorithms 1 and 2 show the pseudo-code for the general procedure for converting a select-project query on a single DPM-based view into a query over the particle table. Given an SQL query Q over a DPM-based view, Algorithm 1 computes the result query Q' and Algorithm 2 computes the confidence query Q''.

Intuitively, the occurrence of a non-key attribute in the *select* clause is replaced with a weighted average of the same attribute. Any predicates in the *where* clause on non-key attributes are moved to the confidence query, since they can only affect the confidence in the result. Finally, a correlated sub-query is used to ensure that only the tuples with sufficient confidence are returned to the user.

The μ construct (Section 3) is treated as a key attribute for the purpose of query conversion. For instance, if a query contains predicate $\mu(temp) > 5$, we replace it with $sum(temp * weight) > 5$ and keep this predicate in the result query. If a view contains a discrete non-key attributes, the final result returned to the user is the most likely value of the attribute (the value with the highest probability). The above algorithms can be extended in a fairly straightforward manner to handle this case as well; we omit the details due to space constraints.

5.2 Aggregate Queries

We now show how to transform probabilistic aggregate queries. Prabhakar et al. [8] propose a classification of aggregate queries that can be posed on an uncertain database. These are broadly classified as *value queries*, that return a single number or aggregate, and *entity queries*, that return a set of objects that satisfy the query. They are further classified as:

• Aggregate Value queries

(1) Probabilistic Sum, Avg Query (VSumQ, VAvgQ)
(2) Probabilistic Min, Max Query (VMinQ, VMaxQ)

• Aggregate Entity queries

(1) Probabilistic Range Query (ERQ)
(2) Probabilistic Min, Max Query (EMinQ, EMaxQ)
(3) Probabilistic Nearest Neighbor Query (ENNQ)

We illustrate our conversion routines for each type of query mentioned above.

Probabilistic AVG query, SUM query (VAvgQ, VSumQ)
Consider an SQL query on the *bbqview* that asks to compute the average temperature over all sensors at time equal to 20. To transform this query to a query on the particle table, we first create a temporary table *TBL* that contains the temperature measured by each sensor individually. We then compute the average of these temperatures in order to compute the average across all sensors. In essence, we are exploiting the linearity of expectation to compute the average. The transformed query is shown in Figure 10(i). The transformation process for the Probabilistic Sum Query (VSumQ) is similar. However, other aggregates such as MIN, MAX and nearest neighbor (NN) do not have such properties and in general we need much more complicated SQL queries in order to compute those aggregates. We will now consider the MIN aggregate.

Probabilistic Min Query (VMinQ, EMinQ)
An example of the *entity* version of a MIN query (EMinQ) is a query that asks for the sensor recording the minimum temperature at time equal to 20, whereas the *value* version asks for the minimum temperature itself. Since temperature is an uncertain attribute, each sensor has a (potentially infinite) set of likely temperature values that it can take. In general, the ranges of the values among sensors overlap and hence every sensor might have candidate values that could potentially be the overall minimum temperature. Equivalently, every sensor might have a certain probability of being the one recording minimum temperature.

To answer this query, we use the weighted samples to, in essence, do *numerical integration* over a complex multi-variate integral [8]. For each of the samples for a sensor S, we compute the probability that it is the minimum value. This is done by constructing two temporary tables TBL1 and TBL2 as shown in Figure 10(ii).

Intuitively, to compute the probability that a particular value T_i for sensor S is a candidate minimum, we compute the probability that each of the *other* sensors measure a temperature greater than T_i and multiply these quantities. This involves performing a self-join of the particle table and then computing the sum of weights of tuples in each of the other sensors that have temperature values greater than T_i and then multiply the sums. Since there are no operators to multiply row values within a Group-by clause (we can only compute sums), we evaluate the necessary product by turning it into a sum using logarithms. The above information is stored in TBL1. Table TBL2 is constructed by pruning the sensors that have no probability of being the minimum. This check is performed using the Having clause in CREATE TABLE TBL2 statement. After computing tables TBL1 and TBL2, we compute the results for the Entity query and the Value query as shown in the Figure.

Probabilistic Max Query (VMaxQ, EMaxQ)
Transformation for the Probabilistic Max queries (VmaxQ, EMaxQ) is very similar to that for the Min query (VMinQ, EMinQ) respectively. The only difference is that, instead of using the $<$ comparison operator to create the temporary table TBL1, we use the $>$ operator. The rest of the query is the same as before.

```
WITH TBL AS ( SELECT id, sum(weight*temp)
              FROM particles WHERE time = 20
              GROUP BY id )
SELECT avg(temp) FROM TBL T;
```

(i) Transforming the AVG query (Original Query not shown)

```
WITH TBL1 AS (
 SELECT    p.id AS pid, p.temp AS temp,
           p.weight AS weight, q.id AS qid
           log(sum(q.weight)) AS logsum
 FROM      particles p, particles q
 WHERE     pid != qid AND p.temp < q.temp
           AND p.time = 20 AND q.time = 20
 GROUP BY  p.id, p.temp, p.weight,q.id
)
WITH TBL2 AS (
 SELECT    pid, temp, weight,
           exp(sum(logsum)) AS probability
 FROM      TBL1
 GROUP BY  pid, temp
 HAVING    Count(*)= (SELECT COUNT(distinct id)
           FROM particles) - 1
)
SELECT sum(probability*weight*temp)
FROM TBL2;
                 (a) VMinQ
SELECT pid, sum(probability*weight)
FROM TBL2 GROUP BY pid;
                 (b) EMinQ
```

(ii) Transformation of MIN query (Original Query not shown)

Figure 10: Examples of Aggregate Query Transformation.

Probabilistic Range Query (ERQ)

Consider an SQL query that determines the sensors that measure temperatures between a given range $[T_l, T_u]$. As temperature is an uncertain attribute, every sensor will have a non-zero probability of measuring temperature in the above range. Therefore, to execute this query, we need to integrate the temperature probability distribution of each sensor in the given range $[T_l, T_u]$ to compute the necessary probability values. Since, we have a sample based representation of the temperature probability distribution, the integration can be easily performed by just computing the sum of weights. The transformed query on the particle table is shown in Figure 11(i).

Probabilistic Nearest Neighbor Query ENNQ

```
SELECT id, sum(weight) FROM particles
WHERE time = t and temp > T_l and temp < T_u
GROUP BY id, time
```

(i) Transformation of the Probabilistic Range query (Original query not shown)

```
WITH TBL1 AS (
 SELECT    p.id AS pid, p.temp AS temp,
           p.weight AS weight, q.id AS qid
           log(sum(q.weight)) AS logsum
 FROM      particles p, particles q
 WHERE     pid != qid AND abs(p.temp-r) < (q.temp-r)
           AND p.time = 20 AND q.time = 20
 GROUP BY  p.id, p.temp, p.weight,q.id
)
WITH TBL2 AS (
 SELECT    pid, temp, weight,
           exp(sum(logsum)) AS probability
 FROM      TBL1
 GROUP BY  pid, temp
 HAVING    Count(*)= (SELECT COUNT(distinct id)
           FROM particles) - 1
)
SELECT sum(probability*weight*temp)
FROM TBL2;
               (a) VMinQ
SELECT pid, sum(probability*weight)
FROM TBL2 GROUP BY pid;
               (b) EMinQ
```

(ii) Transformation of the Nearest Neighbor query (Original Query not shown)

Figure 11: More examples of Aggregate Query conversion

An example of the nearest neighbor query is the following. Report the sensor(s) that measures temperature closest to a given value r at time t. In essence, we need to determine the sensor i such that $\forall_{j \neq i} |T_i - r| < |T_j - r|$. Effectively, we are determining the sensor with the minimum value of $|T - q|$. We can therefore execute the Nearest neighbour query just as the Min query, only with a different expression to minimize. The transformed query is shown in Figure 11(ii).

6 Update Manager

The update manager is in charge of keeping the particle table updated and consistent with the incoming data stream. We use a sequential Monte Carlo technique called *particle filtering* for this purpose. We present a brief overview of this inference technique next.

6.1 Particle Filters

Particle filtering [14] is a well known sequential Monte Carlo algorithm for performing state estimation in DPMs, and has been shown to be effective in a wide variety of scenarios. In short, the algorithm computes and constantly maintains sets of *particles* to describe the historical and present states of the model. As discussed in Section 3, this is essentially the internal representation that our system uses to maintain the DPM-based views.

There are five components to the particle filtering technique: *initialization*, *prediction*, *filtering*, *re-sampling*, and *smoothing*. Algorithms 3, 4 and 5 show the pseudo-code for some of these routines. We illustrate thse routines using the BBQ DPM(Figure 3(i)).

Initialization: At the beginning of the process, an initial set of particles is created by randomly sampling from the prior distributions on the attributes.

Prediction: The prediction step is invoked to advance *time*. During this step, the state at time $t + 1$ is estimated using the state at time t. This is done using the conditional probability distributions associated with the model. More specifically, for each existing particle at time t, a new particle for time $t + 1$ is created by sampling from an appropriate CPD. Let (T_t^i, H_t^i) denote the ith particle at time t for our model. The corresponding particle at time $t+1$, (T_{t+1}^i, H_{t+1}^i), is created by sampling from the distributions $p(T_{t+1}|T_t, hour_{t+1})$ and $p(H_{t+1}|H_t, hour_{t+1})$ where $hour_{t+1}$ is the hour at time $t + 1$.(Section 2.3)

Filtering: The filtering procedure involves using the updates that arrive at time $t + 1$ to update the state estimate at time $t + 1$. First, each new particle is assigned a weight based on the values of the observed variables at time $t + 1$. Particles closer to the observed values receive higher weights compared to the particles that are further from the observed values. These weights are computed using the CPDs of the observed nodes. For our running example, the weights are assigned to the predicted particles based on the CPD of the observed node M_t, $p(M_t|H_t)$. This step is typically integrated with the prediction step (Algorithm 3). At the end of the step, the weights are normalized so they sum up to 1.

Re-sampling: One of the critical problems with the particle filtering technique is that it may degenerate to the case where a single particle has all the weight. This is handled through a re-sampling step, where the set of particles created in the filtering step are re-sampled to generate a new set of particles. The re-sampling step creates a new set of particles, all with the same weight, thus taking care of the degeneracy. Note that the same particle may be repeated multiple times in the resulting set of particles. This is not a problem as the next prediction step will generate different new particles from these identical particles.

Smoothing: This routine uses the current state distribution to "correct" the state at previous times. We illustrate this with an example. Consider a scenario where the temperature being modeled changes suddenly. However, the first reading that contains this change may not affect the inferred temperature because of the noise in the process. Over time as new readings arrive all confirming the change, the inference process becomes more sure of the change in temperature. Intuitively speaking, the earlier change that was attributed to the noise, should now be re-attributed to an actual change in the temperature. This is done using the smoothing procedure which recomputes the weights of the particles at earlier times. In theory, the change should be propagated all the way back to $time = 0$. However the effect typically diminishes after a few steps, and we only update the distribution of those steps that are atmost L time units away (where L is called the *smoothing lag*). The pseudo-code for smoothing is shown in Algorithm 4.
The Smoothing step is present in order to reduce the variance in the filtering output. However, it is a very expensive operation - $O(N^2L)$ where N is the number of particles; and hence not performed at every time step. This offers trade-off between accuracy and performance where in we can control the smoothing oper-

Algorithm 3 Prediction and Filtering

Require: Particles at time t; Observations obs made at time $t+1$.

1: **for** each particle p_t^i at time t **do**
2: Create a partial particle, p_{t+1}^i, for time $t+1$
3: **for** every hidden node X_{t+1}^j in the time slice for $t+1$ in the topological order **do**
4: Let $Pa(X_{t+1}^j)$ denote the parents of X_{t+1}^j
5: Let v denote the values of $Pa(X_j)$ (from p_t^i or from p_{t+1}^i constructed so far)
6: Sample a value for X_{t+1}^j from $p(X_{t+1}^j|Pa(X_t^j) = v)$.
7: Add the new sampled value to p_{t+1}^i.
8: Set weight of new particle, w_{t+1}^i = 1.
9: **for** every observed node Y_{t+1}^j **do**
10: w_{t+1}^i = $w_{t+1}^i\, p(Y_{t+1}^j = obs_j|Pa(Y_{t+1}^j))$

Algorithm 4 Smoothing

Require: Smoothing lag L; Particles from time T to $T-L$

1: **for** time t = T to T-L **do**
2: **for** each particle p_t^i at time t **do**
3: $w_t^i = w_t^i \sum_{j=1}^N w_{t+1}^j \frac{p(X_{t+1}^j|X_t^i)}{\sum_{k=1}^N w_t^k\; p(X_{t+1}^j|X_t^k)}$

Algorithm 5 Update Manager

1: Initialize the particle table by *sampling* from the prior distributions of the hidden nodes in the first slice of DPM.
2: **for** each time instant t **do**
3: *Predict* a new set of particles for time $t+1$ from the old particle set at time t.
4: **if** New data obs_t in data stream at time t **then**
5: *Filter* particles based on obs_t.
6: *Resample* new particles.
7: *Smooth* the particles at previous times (up to $t-L$) using obs_t.

ation and its lag in order to obtain necessary bounds. But in many models, Smoothing does not significantly improve upon the results (reduce variance) of the filtering step and in those cases filtering alone guarantees sufficient modeling accuracy with good performance.
The update manager repeatedly invokes the routines described above to keep the particle table updated as new data tuples arrive. The pseudo-code for the update manager is shown in Algorithm 5.

7 Learning

7.1 Maximum Likelihood Estimation

Maximum likelihood estimation (MLE) is a popular statistical method used to learn parameters of the underlying probability distribution from a given data set. Suppose that we have a sample of data $X_1, \ldots, X_n$ and we want to infer the distribution it came from. A common assumption made is that the data is independent and identically distributed (iid) from a parametric distribution with unknown parameters. We use

the MLE technique to estimate the unknown parameters. If the probability distribution to be determined has parametric form f and is parametrized by θ, then we try to maximize the likelihood function, which is given by.

$$L(\theta) = f(x_1, x_2, x_3 \ldots, x_n|\theta)$$

Since the data are iid, we can simplify the above expression as follows.

$$\begin{aligned} L(\theta) &= \pi_i^n f(x_i|\theta) \\ log(L(\theta)) &= \sum_i^n log(f(x_i|\theta)) \end{aligned}$$

We then determine the value of θ that maximizes the above log likelihood function. A simple illustration of the above method for determining parameters of the Gaussian distribution is shown below.

$$\begin{aligned} p(x_i|\mu, \sigma) = N(\mu, \sigma) &= \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{\sigma^2}} \\ \text{Hence,} \quad log(L(\mu, \sigma)) &= \sum_{i=1}^n \left(-log(\sigma) - \frac{(x_i-\mu)^2}{\sigma^2} \right) \end{aligned}$$

Optimizing the above expression with respect to μ and σ independently gives us the following values.

$$\begin{aligned} \mu &= \sum_{i=1}^n x_i/n \\ \sigma &= \sum_{i=1}^n (x_i-\mu)^2/n \end{aligned}$$

7.2 Learning CPDs

For the applications in our system, we have to learn conditional probability distributions of the form $P[X|Pa(X)]$. In order to learn CPDs, we initially learn the joint distribution $P[X, Pa(X)]$ using MLE and then obtain the expression for the conditional distribution using Baye's theorem. We continue to illustrate using the Gaussian example. Let us suppose we want to compute $P[X|Y]$ from $P[X, Y]$. Let Z denote the random variable $[X, Y]$.

Let the vector $z = [X^T, Y^T]^T$ be distributed according to:

$$z = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} \sim N\left(\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}, \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C^T} & \mathbf{B} \end{bmatrix} \right)$$

Then, the conditional distribution $Pr[X|Y]$ (which is computed using Baye's rule) is given by

$$P[\mathbf{X}|\mathbf{Y}] = N(\mathbf{a} + \mathbf{cB^{-1}}(\mathbf{y} - \mathbf{b}), \mathbf{A} - \mathbf{CB^{-1}C^T})$$

8 Implementation Details

In this section, we briefly discuss the implementation details of our system. We have built a prototype of our system using Java, and we use the open source Apache Derby (Java embedded database system) [3] to store

```
(i)  SELECT first.OID, second.OID, first.time
     FROM kfview first, kfview second
     WHERE (first.time = second.time)
     AND (|first.μ(x) - second.μ(x)| < δ)
     AND (|first.μ(y) - second.μ(y)| < δ)

(ii) SELECT kfview.x, kfview.y
     FROM kfview WHERE kfview.OID = 4
```

Figure 12: Queries used in the experiments: (i) An intersection query, and (ii) a trajectory query on the KFM-based view for moving objects;

the particle tables. Our prototype implementation is currently an application level software that lies above the Derby abstraction layer. The application accesses the particle tables using JDBC calls. In addition, we cache the particles that belong to the last L time steps (smoothing lag, Section 6.1) in memory for efficient access; the particles are written to the database in background. We are currently working on moving the entire implementation inside Derby. The experiments we present in the next section were conducted on a Linux machine with 1.8 GHz Intel Pentium-IV processor with 512 MB of memory.

Perhaps the single most important challenge we faced in our implementation was handling the many different possible types of node variables and their associated CPDs. Nodes in a DPM can be continuous or discrete (with a variety of domains). A node may have any number of discrete or continuous parent attributes, and the CPD itself may be a known parametric function or may be non-parametric. Hence the implementation needs to be generic in order to support the various forms. Also, we need to provide flexibility to the user to easily augment the system, for instance, by adding a new CPD that is not currently supported but is suitable for her application.

Extensible CPD API

We abstract all the details specific to CPDs inside a CPD object and export only the generic functions that are necessary for the higher level learning and inference procedures. To add a new probability distribution for use as a CPD, the implementor must provide the following functions:

- **public Object getSampleFromCPD(ArrayList pVals):**
  This function produces a new sample value for the node given the value of its parents (supplied in the ArrayList).
- **double getProbability(double val, ArrayList pVals):**
  This function returns the probability that the node variable takes the value **val**, given its parents values (as specified in **pVals**).
- **public void addSample(double val, ArrayList pVals):** This function adds a new data sample to the repository of samples used to learn this particular CPD.
- **public void computeParams():** This function, invoked after all the training samples are added, is used to "learn" the parameters of the CPD.

9 System Evaluation

In this section we present results from the experimental evaluation of our prototype implementation. The salient points of our study can be summarized as follows:

1. **Necessity:** Using dynamic probabilistic models is a must to deal with erroneous and incomplete real-world data streams.
2. **Accuracy:** Error (MSE) obtained in the inference process follows theoretically expected $1/N$ behavior [14] where N is the number of particles. With as few as 100 particles, the error percentage obtained on a real dataset was less that 1%.
3. **Efficiency:** Inference times required by our system, even with large number of particles (about 1000) is quite small (less than 20ms) and it is hence feasible to use it for online inference in high rate data streams.

9.1 Experimental setup

We use the following two datasets for our evaluation.

Dataset I: Moving Objects

With an increasing amount of location data (e.g. GPS data) available today and the interest in providing location-based services using such data, moving objects databases have received much attention in recent years [37, 42, 8]. We consider a moving objects scenario where a number of point objects are moving around in a 2-dimensional space. We assume that each object is fitted with a GPS device that constantly transmits its position to a central server. The GPS data may contain noise, and some location updates may be lost in the process. We are interested in modeling the above data stream to infer the true locations and the velocities of the objects. Lacking a real-world dataset with GPS traces over multiple objects, we instead generate simulated data with the properties described above. We simulate GPS readings of random trajectories for these objects. We add a white Gaussian noise with a standard deviation of 2 units to insert noise in the data. In addition, we drop 5% of the readings randomly to model incomplete data and communication failures.

We use the KFM model to process this data and to infer the true locations and velocities (Section 2.2). We perform smoothing operation with a lag of 2. Each moving object is modeled separately using a different KFM model, but we store the information about all objects in a single table. We use the view creation command shown along with the configuration file shown in Figure 7(i) to create these views. The schema of this view is:

$$kfview(time, OID, x, y, v_x, v_y)$$

Dataset II: Sensor Data

There has been much work recently [12, 34, 8] on managing noisy and incomplete sensor data and inferring useful information from them. We attempt to use our system to perform similar tasks. We use the publicly available Intel Lab dataset [27] that consists of traces from a 54-node sensor network deployment at the Intel lab in Berkeley. The dataset consists of light, humidity and temperature readings collected in the lab. The readings collected are extremely noisy and in many cases incomplete. Also, several sensors failed midway through deployment and most of them continued to transmit erroneous values leading to more errors. It is also observed the dataset has **intra-tuple correlations** (temperature and humidity) and **inter-tuple correlations** across time. In order to decrease acquisition costs for query processing in power aware sensor networks, the efficient strategy [12] is to observe only a few subset of attributes that are easy to acquire, and infer the rest of the attributes from a probabilistic model. In our experiment we attempt to accurately infer the values of the temperature based on the observed humidity readings using the dynamic probabilistic model shown in Figure 3(i). We ran a series of processings tasks overs this data.

Step 1: Remove Incorrect Data We first remove incorrect values from the dataset. This involves detection

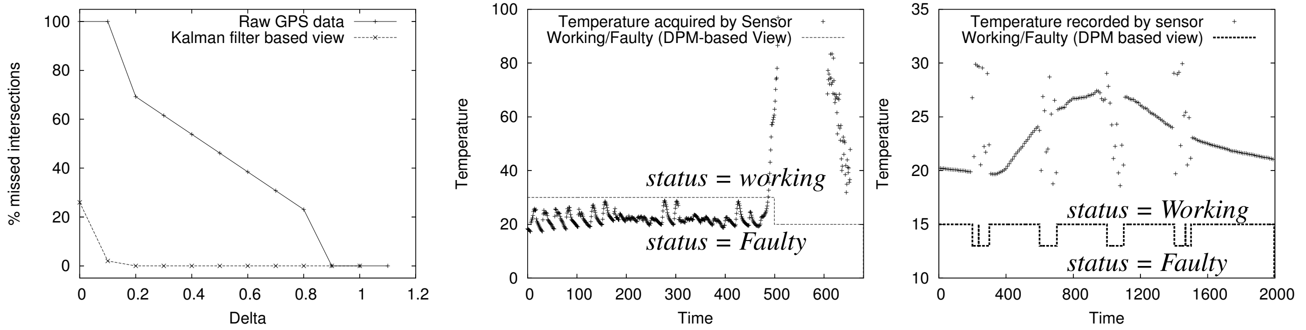


Figure 13: (i) The % of missed intersections as a function of δ on the raw data and the KFM-based view; (ii) The observed temperatures at a sensor, and the *status* inferred using an HMM. (iii) Same as (ii) with simulated faults inserted.

of the failure times of the sensor nodes. We perform this using an HMM-based view based on the HMM model shown in Figure 2(i).

Step 2: Learn DPM Once the incorrect sensor readings are removed, we split the data into training and testing datasets. The training dataset (data collected for 6 days) is used to learn the conditional probability distributions of the nodes.

Step 3: Inferring Temperature values We then use the humidity readings in the test dataset (data collected for 3 days) to infer the temperatures using the BBQ DPM shown in Figure 3(i) by generating *bbqview*.

Step 4: True Temperature Values In order to evaluate the accuracy of the inferred temperatures, we remove noise from the observed temperatures using another DPM based view. (The DPM used is similar to a KFM and is not shown). The resulting correct temperatures are compared with the temperatures inferred from Step 3 to evaluate the accuracy of the inferred temperatures.

9.2 Experimental Results

Applying DPMs to data is critical

Dataset I: To show the importance of modeling the data streams, we pose an *intersection* query over the moving objects database (Figure 12(i)). This query measures the number of intersections between objects, ie., the times at which two particles are closer than a specified distance δ. We execute the same query on the raw GPS data and compare the number of correct intersections that are measured in both cases. Figure 13(i) shows the plot comparing the percentage of missed intersections in the raw data and the KFM-based view. As we can see, a large number of the intersections are missed while executing the query on the raw data, especially for smaller values of δ. This is both because of the missing data and the noise in the data. The KFM-based view, on the other hand, is able to capture most of the real intersections in the data.

Dataset II: Figures 13(ii), (iii) show the temperature values that were measured by the sensors and those readings that were marked as incorrect by the HMM-based view. As we can see from Figure 13(ii), there are several incorrect values after 500 hours (20 days approx), that need to be removed before we can use them for learning. We also added a few simulated faults in order to further verify that the HMM-based view correctly identifies the faulty readings.

Inference using particle filtering is accurate

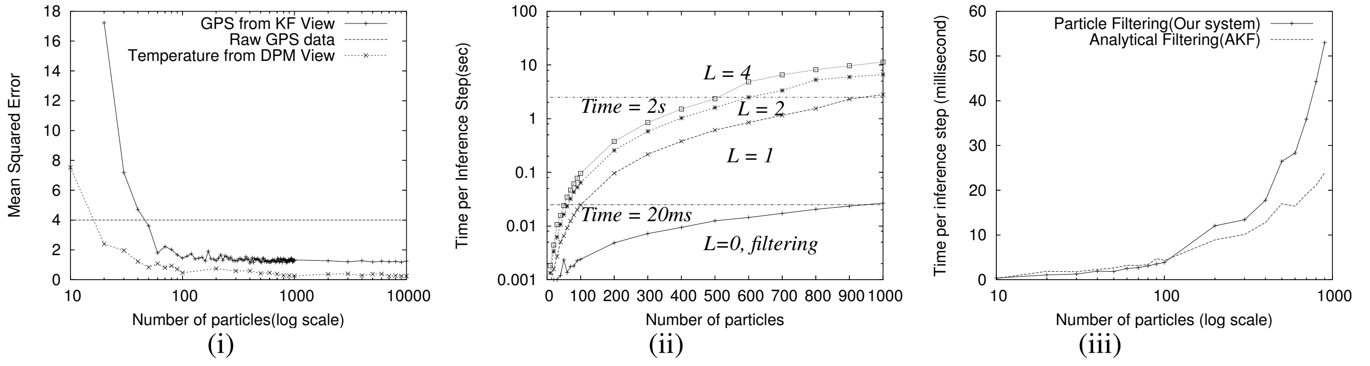


Figure 14: (i) Plot of mean-squared error vs number of particles for both *kfview* (Dataset I) and *bbqview* (Dataset II). Mean squared error falls off as (1/N) (ii) Graph shows time taken for one inference step at various smoothing lags. (iii) The update times for particle filtering are comparable to those for an analytical technique.

With this set of experiments, we show that the particle filtering algorithm accurately determines the hidden state of the system being modeled.
Dataset I: We execute the trajectory query shown in Figure 12(ii), that returns the path traced by object 4, on the raw data and on the KFM-based view. The accuracy of the result is measured by computing the *deviation* of the path from its *actual* path using the sum-squared error function. We plot the estimate of the error as a function of the number of particles (N) in Figure 14(i).
Dataset II: We compare the value of temperatures that were inferred by the *bbqview* (with just filtering, no smoothing) to the true temperature values generated in Step 4. We compute a mean square error estimate and plot the mean squared error as a function of the number of particles. We obtain the graph shown in figure 14(i).

From the plots, we can see that the error in the KFM-based views for GPS datasets is much less than that in the raw data. (Error in raw data is indicated by the straight line.) We also verify that the error decreases as the number of particles used increases. For low values of N, the error reduces drastically in the beginning, however, for higher values of N (more than 100 particles), it remains fairly constant. Both the error graphs follow the theoretically estimated $(1/N)$ graph which validates our experiments. For the temperature data (dataset II), the mean square error obtained on the test data with just Filtering alone is less than 0.25 units ($\leq$1% error) when just 100 particles are used.

Inference using particle filtering is efficient

Next, we provide details regarding the performance of our system.
Learning: Given data to be modeled and a DPM, time is initially spent for learning the CPDs. Learning the CPDs for the Temperature and Humidity nodes in the BBQ DPM from about 430000 records (each of dimension 3) took about 7.5 seconds.

Inference: After the CPDs are learnt and we receive data continuously, time is spent on performing the Inference procedure. The inference procedure, performed at each time instant, results in addition of several new rows and modification of already existing rows, if smoothing is used. We measure the time taken for one inference step as a function of the number of particles. We carry out this experiment for different values of the smoothing lag parameter, $L = 0, 1, 2, 4$. The results obtained are shown in Figure 14(ii). We find that the execution time increases linearly with increase in the number of particles (as y-axis is in log scale, this cannot be explicitly seen). If we perform only filtering, the inference time is very small; we process

more than 1000 particles in just 20ms. However, if we continuously perform smoothing, the time taken for inference increases drastically as shown in the graph. However, even with a smoothing lag of 4 time steps, we can process 100 particles in less than 100ms (reasonable for most common streams). As the accuracy graph shows in Figure 14(i), this may be enough to achieve sufficient accuracy. We are considering “lazy” smoothing strategies where we perform smoothing ocassionally (not at every time step) and only when it is essential.

Comparison to Analytical Filtering: As there are no other systems that perform tasks similar to our system, we implemented an analytical solution for Kalman Filters (called *AKF*). We compare the performance of our particle filtering inference algorithm against AKF. In AKF, particles are directly sampled using the mathematical equations for the Kalman filter [43] and are inserted into the database in order to obtain a common ground for comparison. Note that even for AKF, we have to generate the samples in order to execute queries against the output of the model. We compare the time taken by our system (the KFM-based view for Dataset I) against AKF. For both the systems, we do not perform the Smoothing procedure. Figure 14(iii) shows the execution times for a single inference step as a function of the number of particles. As expected, the Kalman filter implementation using analytical formulas takes less time than the particle filtering algorithm. However the difference between the two is not significant except when the number of particles is very large. We would also like to note that our prototype implementation is not fine-tuned for performance, and we expect that a more careful implementation will reduce the time taken by our implementation even further.

10 Related Work

Bayesian Networks and Dynamic Probabilistic Models: Bayesian networks have a long and rich history across a wide variety of research disciplines, and numerous books have been written addressing several aspects of such models (e.g. [35, 22]). Dynamic probabilistic models (sometimes called dynamic Bayesian networks, or dynamic probabilistic networks), a relatively newer and less well-established concept, allow reasoning about the temporal dimension as well, and are used extensively for modeling complex stochastic processes [16, 33, 30]. In recent years, they have been seen as the tool to unify similar concepts in seemingly disparate domains such as probabilistic expert systems, statistical physics, image analysis, genetics and so on [41, 18]. For instance, both HMMs and Kalman filters, perhaps the most common examples of such models, were developed independently in engineering and speech recognition communities, and their similarities to graphical models have been observed fairly recently [40, 25]. Over the years, several general purpose toolkits have been developed that support Bayesian networks, and in some cases, dynamic Bayesian networks (e.g. [32, 20]). However, to the best of our knowledge, ours is the first work that proposes to directly implement arbitrary dynamic probabilistic models inside databases thereby making it easier to use DPMs, and also increasing the functionality and appeal of relational database systems. Probabilistic relational models [17] are a generalization of Bayesian networks to learning probabilistic models of the data present in relational tables. More recently, dynamic probabilistic relational probabilistic models have also been considered in the machine learning community [38]. This work is also complementary to our approach here.

Probabilistic Databases: In recent years, we have seen a renewed interest in the area of probabilistic databases, fueled primarily by a large increase in the amount of real-world data that is inherently noisy, incomplete and uncertain (e.g., [24, 4, 11, 44, 39]). Several research efforts are underway to build systems to manage uncertain data (e.g. MYSTIQ [11], Trio [44], ORION [8], ConQuer [2]). As we discussed in Section 3, views based on dynamic probabilistic models are naturally probabilistic, and we plan to use the

techniques developed in the probabilistic databases research, especially query languages and semantics, in our future work.

Data Streams & Sensor Networks: Many data stream management systems have been proposed for real-time processing of continuously generated data by sensor networks [6, 7, 5, 31]. The main focus of that work has been on efficient evaluation of large number of continuous queries over high-rate streaming data. Our work is complementary to this work; it focuses on efficiently modeling streaming data so that the query results can be more meaningful and useful to the user.

Sensor networks have been a very active area of research in recent years (see [1] for a survey), and there is a large body of work on data collection from sensor networks that applies higher-level techniques to sensor network data processing. TinyDB and Cougar [45, 28] provide declarative interfaces to acquiring data from sensor networks. Several systems propose to use probabilistic modeling techniques to answer queries over sensor networks [21, 12, 10], though these have typically used specific models rather than a generalized implementation in an existing relational database. Finally, as discussed in Section 3, our system generalizes and significantly enriches the abstraction of model-based views that was originally proposed in the MauveDB system [13].

11 Conclusions

Advances in miniaturization technology and networking have resulted in a rapid increase in the number of large-scale deployments of measurement infrastructures that continuously generate tremendous volumes of priceless data. In this paper, we presented an approach to build an extensible database system that enables users to apply general purpose dynamic probabilistic models to such data in real-time, thus significantly enriching the functionality and the appeal of databases for managing such data. We provide intuitive interfaces to declaratively specify the models to be applied, and to query the output of the application of such models to data streams. We use particle filtering to perform the inference tasks, and we show how this also enables efficient query execution over DPM-based views. The techniques we develop for representing and querying probabilistic tables using particles are of independent interest to the probabilistic database community as well. Our experimental evaluation over a prototype implementation illustrates the advantages of enabling real-time application of dynamic probabilistic models to streaming data.

References

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38, 2002.

[2] Periklis Andritsos, Ariel Fuxman, and Renee J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.

[3] The Apache Derby Project. Web Site. `http://db.apache.org/derby/`.

[4] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE TKDE*, 4(5):487–502, 1992.

[5] D. Carney. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.

[6] Sirish Chandrasekaran. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[7] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[8] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.

[9] Tanzeem Choudhury, Matthai Philipose, Danny Wyatt, and Jonathan Lester. Towards activity databases: Using sensors and statistical models to summarize people's lives. *IEEE Data Eng. Bull.*, 29(1):49–58, 2006.

[10] M. Chu, H. Haussecker, and F. Zhao. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. In *Intl Journal of High Performance Computing Applications*, 2002.

[11] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

[12] Amol Deshpande, Carlos Guestrin, Sam Madden, Joe Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[13] Amol Deshpande and Samuel Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD Conference*, pages 73–84, 2006.

[14] Arnaud Doucet, Nando de Freitas, and Neil Gordon. *Sequential Monte Carlo methods in practice*. Springer, 2005.

[15] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge Univ Press, 1999.

[16] N. Friedman, K. Murphy, and S. Russell. Learning the structure of dynamic probabilistic networks. In *UAI*, 1998.

[17] Lise Getoor. *Learning Statistical Models from Relat- ional Data*. PhD thesis, Stanford University, 2001.

[18] Zoubin Ghahramani. Learning dynamic Bayesian networks. *Lecture Notes in Computer Science*, 1387, 1998.

[19] E. Hoke, J. Sun, and C. Faloutsos. Intemon: Intelligent system monitoring on large clusters. In *VLDB*, 2006.

[20] S. Hjsgaard and C. Dethlefsen. The grbase package for graphical modelling in R. Technical Report R-2004-19, Aalborg University, 2004.

[21] A. Jain, E. Change, and Y. Wang. Adaptive stream resource management using kalman filters. In *SIGMOD*, 2004.

[22] Michael I. Jordan. *Learning in Graphical Models (ed)*. MIT Press, 1998.

[23] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli. On-line fault detection of sensor measurements. *IEEE Sensors*, 2003.

[24] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *ACM TODS*, 22(3), 1997.

[25] B.C. Levy, A. Benveniste, and R. Nikoukhah. High-level primitives for recursive maximum likelihood estimation. *IEEE Trans. on Automatic Control*, AC-41(8), 1996.

[26] D. Lymberopoulos, A.S. Ogale, A. Savvides, and Y. Aloimonos. A sensory grammar for inferring behaviors in sensor networks. In *IPSN*, 2006.

[27] Sam Madden. Intel lab data, 2004. `http://berkeley.intel-research.net/labdata`.

[28] Samuel Madden, Wei Hong, Joseph M. Hellerstein, and Michael Franklin. TinyDB web page. `http://telegraph.cs.berkeley.edu/tinydb`.

[29] A. M. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, 2002.

[30] V. Mihajlovic and M. Petkovic. Dynamic bayesian networks: A state of the art. University of Twente Document Repository 2001.

[31] Rajeev Motwani. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.

[32] Kevin Murphy. The Bayes net toolbox for matlab. *Computing Science and Statistics*, 33, 2001.

[33] Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learnig*. PhD thesis, UC Berkeley, 2002.

[34] D. Patterson, L. Liao, D. Fox, and H. Kautz. Inferring high level behavior from low level sensors. In *UBICOMP*, 2003.

[35] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[36] L Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. 77:257–286, 1989.

[37] K. Raptopoulou, M. Vassilakopoulos, and Y. Manolopoulos. Towards quadtree-based moving objects databases. *Lecture Notes in Computer Science, Volume 3255, Jan 2004*, 2004.

[38] S. Sanghai, P. Domingos, and D. Weld. Dynamic probabilistic relational models. In *IJCAI*, 2003.

[39] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.

[40] P. Smyth, D. Heckerman, and M. Jordan. Probabilistic independence networks for hidden markov models. *Neural Computation*, 9(2):227–269, 1997.

[41] Padhraic Smyth. Belief networks, hidden markov models, and markov random fields: a unifying view. *Pattern Recognition Letters*, 18(11-13), 1997.

[42] Goce Trajcevski, Ouri Wolfson, Klaus Hinrichs, and Sam Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, 2004.

[43] Greg Welch and Gary Bishop. An introduction to the Kalman filter. `http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html`, 2002.

[44] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

[45] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.

[46] Jie Ying. A hidden markov model-based algorithm for fault diagnosis with partial and imperfect tests. *IEEE Trans. on Systems, Man, and Cybernetics, Part C*, 2000.