Online Filtering, Smoothing and Probabilistic Modeling of Streaming data

Bhargav Kanagal #1,Amol Deshpande #2

University of Maryland

[1]bhargav@cs.umd.edu

[2]amol@cs.umd.edu

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*, to represent the present and historical states of the model as sets of weighted samples (*particles*) that are kept up-to-date as new data arrives. 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 several synthetic and real datasets, that demonstrates the feasibility of online modeling of streaming data using our system and establishes the advantages of tight integration between dynamic probabilistic models and databases.**

I. 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 [22] to behavior of large computational clusters [14]. 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 [5], [6], [4], [24] that can process high-rate data streams in real-time and can 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 [11]). A common task over such data streams is to continuously *infer* the value of the hidden variables using the observed data. Hidden Markov models [29], or variations thereof, are often used for this purpose; these 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 II-A).
- **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), 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 [33] and its extensions 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* [8], [27], [20]. For instance, Patterson et al. [27] 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 in the data and so on, that cannot be expressed as SQL queries. Because of this, the majority of the analysis and querying in these applications is typically done outside the database system, leading to much repetition of functionality and highly inefficient execution.

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 stream-

ing data [26], [23]. We use the recently proposed abstraction of *model-based views* [12] to push the application of a wide range of DPMs to streaming data inside a relational DBMS, thus enabling easy application of these tasks. By exploiting the structure of *particle filters* (a widely applicable *sequential Monte Carlo* technique), we efficiently implement DPMs and represent them as sets of weighted samples (called *particles*) in relational tables. This representation of DPMs naturally captures many of the attribute correlations present in the data. In addition, we have designed novel techniques to rewrite queries posed over a single DPM-based view (including aggregate queries) into queries over the above representation, allowing us to exploit the existing query processing machinery. Our experimental results reveal that our system achieves sufficient accuracy (99%) in modeling data streams and that the processing times are quite reasonable (20ms/tuple); we can easily handle streams upto 50 tuples/sec. GPS and many common streams have rates much below this figure (< 1 tuple/sec).

The rest of the paper is organized as follows. We begin with an overview of dynamic probabilistic models in Section II. We describe the abstraction of DPM-based views in Section III and present a detailed description of our system and the algorithms used in Section IV-V. We conclude with an experimental evaluation of our prototype implementation in Section VI.

II. Background - DPMs

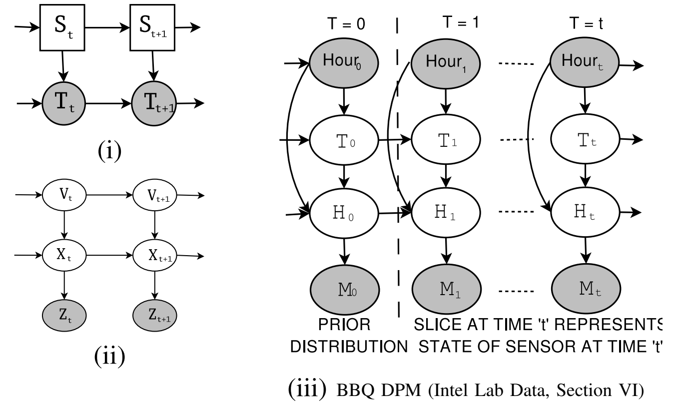
Dynamic probabilistic models are widely used in practice to model and to reason about complex real-world stochastic processes [16], [26], [23]. The simplest and most widely used examples of DPMs are *hidden Markov models (HMMs)* and *linear dynamical systems* (better known as Kalman filter models (KFMs)). We start by illustrating HMMs and then describe more general DPMs. A more detailed illustration of DPMs can be found in the extended version of the paper [17].

A. Hidden Markov models (HMMs)

HMMs have been applied in a variety of areas like speech recognition, bioinformatics, and fault detection [29], [18], [35]. They are used to *infer* the values of unobservable (hidden) state variables from imprecise observations about related variables. We illustrate HMMs using a fault detection application. Consider a single sensor, possibly faulty, that is measuring temperatures in a room and transmitting them to a central database server. We wish to know if the sensor is working correctly (so we can ignore erroneous readings). The only information we have about the sensor are the temperature readings it transmits.

We can use an HMM to solve this problem as follows. The *hidden* variable in this case (which we cannot measure), is the working status of the sensor, S_t, which takes two values: *Working (Wo)* or *Failed (Fa)* (t denotes the time). The *observed readings* are the temperatures, T_t, measured by the device, which may contain noise.

The prior knowledge about the system behavior (that is used to determine whether the sensor has failed) can be captured by two *conditional* probability distributions (Figure 1(iv)) :



$p(T_{t+1}|T_t, S_{t+1} = Wo) = N(T_t, \sigma)$
$p(T_{t+1}|T_t, S_{t+1} = Fa) = U(0, 100)$

$p(S_{t+1}|S_t) =$

	Wo	Fa
Wo	0.99	0.01
Fa	0.01	0.99

(iv) CPDs for HMM

$p(v_{t+1}|v_t) = N(v_t, \sigma_v)$
$p(x_{t+1}|x_t, v_{t+1}) = N(x_t + v_{t+1}, \sigma_x)$
$p(z_{t+1}|x_{t+1}) = N(x_{t+1}, \sigma_y)$
$Priors: \ p(v_0) \ and \ p(x_0)$

(v) CPDs for KFM

Fig. 1: Graphical representations and CPDs for various DPMs: (i),(iv) HMM for fault detection; (ii),(v) KFM for velocity and location estimation (Section VI); (iii) BBQ DPM (Section VI).

- $P(T_{t+1}|T_t, S_{t+1})$**:** This distribution captures the behavior of the sensor based on its working status. For instance, from prior knowledge about the process, we expect that if the sensor is working correctly (S_{t+1}=*Working*), then the sensor temperature measured at time $t+1$ should be around T_t (temperature measured at t) plus a small (Gaussian) noise. If the sensor is faulty, then a simple assumption is that the sensor arbitrarily returns any value between 0 and 100, independent of the real temperature. Clearly, the faulty behavior depends on the nature of the sensor.
- $P(S_{t+1}|S_t)$**:** Figure 1(iv) 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 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 using an HMM inference algorithm like *forward-backward* and the *Vitterbi algorithm* [29], we can infer the best possible estimate of the hidden variables (in our case, the status of the sensor at various times).

We note here that the above model is not suitable for temperature *prediction*, but only for fault detection (since it does not capture the temporal trends in the temperature).

B. DPMs: Graphical Representation

DPMs are represented using a directed graphical structure (Figure 1), where the graph captures the dependencies between the process variables. Figure 1(i) shows the graphical representation for the HMM described above and (ii) shows

a KFM model for velocity and location estimation (used in Experiments, Section VI). The details of the graphical representation are as follows.

Nodes of the graph represent the attributes of the system being modeled (as random variables). In Figure 1(i), the attributes of the system being modeled are the temperature (T_t) and status (S_t) variables. By convention, the observed nodes are shaded while the hidden nodes are clear.

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 Figure 1 (iii).

Edges/CPDs: The directed edges represent "causality". In Figure 1(i), the working status at time t influences the *measured* temperature at time t. The degree of causality is indicated by the *conditional probability distribution function* (CPD) described above. The CPD of node X is indicated by $P(X|Pa(X))$ where $Pa(X)$ denotes the parents of node X. We need three sets of CPDs to fully specify a DPM:

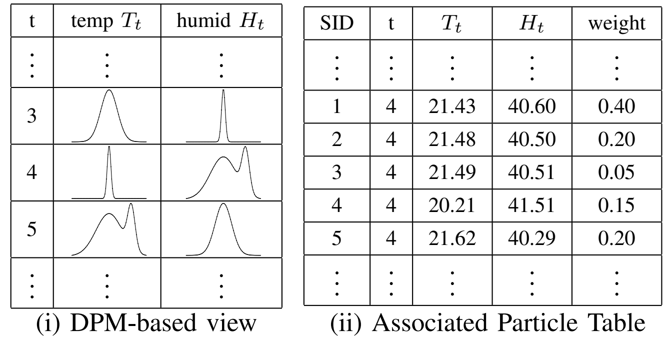
- The prior (unconditional) probability distributions over the variables in the first slice (that may not have any parents).
- The CPDs that encode the knowledge about how the state at time $t+1$ depends on the state at time t.
- The CPDs that encode the knowledge about how the observations at time t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

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 1(i) and (ii)) is usually sufficient. The parameters of the DPM may be input from prior knowledge or may be learned from training data. We use Maximum Likelihood Estimation (MLE) for learning parameters of the CPDs, if needed. Details of the learning algorithm can be found in [17].

BBQ DPM [11]: Figure 1(iii) depicts the DPM that we use as a running example in this paper. Here the observed variables are noisy humidity readings, M_t, and the hour of day, h_t. The hidden variables are true humidity, H_t, and true temperature, T_t, both of which are inferred using M_t; more precisely, at any time t, given the sequence of measurements $M_0, \cdots, M_t$, the DPM can be used to infer probability distributions over values of H_t and T_t. Here, the CPD of node T_{t+1} depends on T_t and the hour of the day h_{t+1} (since how temperature changes depends on the time of the day).

C. Inference in DPMs

The ultimate goal of modeling a stochastic process using a DPM is to obtain a *posterior* distribution over the hidden variables of the 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 IV-C.



t	temp T_t	humid H_t
⋮	⋮	⋮
3		
4		
5		
⋮	⋮	⋮

(i) DPM-based view

SID	t	T_t	H_t	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
⋮	⋮	⋮	⋮	⋮

(ii) Associated Particle Table

Fig. 2: (i) DPM-based views contain probabilistic attributes; (ii) Particle-based representation of the view (only particles corresponding to the second tuple, *time* = 4, are shown for clarity)

III. DPMs as Database Views

The abstraction of *model-based view*, proposed in [12], allows creating database views using statistical models. Examples of model-based views based on non-parametric statistical models like *linear regression* and *interpolation* are described in [12]. Here, we extend this abstraction by allowing views to be defined using DPMs instead. Figure 2(i) shows the schema of the view that could be presented to the user with the BBQ DPM model (Figure 1(iii)). As we can see, the schema contains all the hidden state variables in the DPM as attributes along with a *time* attribute (the observed attribute M may be included as well). It must be noted that we in fact maintain joint distributions (accross all schema attributes) although the figure indicates only marginals (for illustration). As with traditional database views, this is a virtual table that may or may not be *materialized*. Although the above DPM-based view shows only continuous variables, DPM-based views can also have discrete variables. (e.g. *status* attribute in HMM-based view presented to the user in the fault detection example (Section II-A)).

The nature of DPMs forces these to be *probabilistic* views since the attributes of this virtual table may be probabilistic (both T_t and H_t are probabilistic attributes here). The issue of querying and representing such probabilistic data has received much attention in recent years [3], [19], [7], [34], [10], [1], [31], 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. 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 expected values of probabilistic attributes. A predicate such as $\mu(temp) > 30$ indicates that the condition is on the mean value of the temperature attribute.
- $with\ confidence\ c$: This allows the users to specify a minimum confidence in the result tuples returned.

In addition, we support SQL queries with aggregates such as AVG, MIN, MAX and NN (Nearest Neighbor).

DPM-based views exhibit complex and strong attribute correlations that can not be ignored during query processing. Most of the probabilistic databases proposed above either assume 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 (e.g., T_t and H_t above).
- **inter-tuple correlations:** that exist between attributes of different tuples (e.g., T_t and T_{t+1}).

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.

Particle-based representation

We use a representation based on *weighted samples* (called *particles*) to store DPM-based views internally. This not only allows us to handle the complex *continuous* 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 in a separate table called *particle table*. This table is initialized and then constantly updated using the inference algorithm (Section IV-C). The set of particles represents the joint distribution over the attributes in the view. Figure 2(ii) shows a set of particles corresponding to one of the tuples in the view. The schema of 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 2(ii)) as $T_4 = \sum_{i=1}^{N}(T_4^i \times w_4^i)$ = 21.28. Note that, since the particles represent the joint distribution, the intra-tuple correlations are naturally captured in this representation.

The accuracy of this representation depends on the number of particles used (N, a system parameter). It has been shown theoretically that the error in the representation is proportional to $1/N$ [13].

IV. System Design

To model a data stream using an appropriate probabilistic model, the following sequence of steps take place:

1) The user uses the *create view* command to specify the DPM and to create the view (Section IV-A).
2) If the user specifies that the CPDs are to be learned using training data, an MLE-based *learning module* (see [17]) is invoked over the training data.
3) A particle table is created and initialized using the prior distributions (Section IV-C).
4) The particle table is continuously updated by the *Update Manager* in response to the incoming data stream measurements (Section IV-C).

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 IV-B).

Next, we discuss the details of these components in turn.

A. 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 *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>
```

The first line of the statement specifies the schema of the view, including its name and its attributes, just as a traditional database view. The fourth line specifies the data stream to be modeled using an SQL query. The structure and the parameters of the DPM itself are specified using a *configuration file* that is provided with the view definition. Figure 3 shows an example of such configuration files for the HMM presented in Section II. The configuration file consists of:

Properties: of attributes in the DPM – whether they 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 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. This graphical representation is required to be acyclic.

CPDs: Prior and conditional probability distributions (Section II-B) 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 3(i) shows the distributions we currently support. For example, $N(\mu, \sigma)$ represents a normal distribution with mean μ and standard deviation σ. The CPD for node 2 in Figure 3(ii) indicates that, based on the state of node 1 (Wo/Fa), node 3 is either normally distributed with mean 50 and standard deviation 0.05 or uniformly distributed (between 0 and 100). Node 3 of the HMM, in Figure 3(ii) has a discrete distribution that was specified using a transition probability matrix in Figure 1(i).
- *By providing a java module file that supports an appropriate API:* If the probability distribution to be specified

$val(i)$	Variable modeled by node i
$cpd(i)$	CPD of node i
$N(\mu, \sigma)$	Normal distribution with mean μ and variance σ
$U(a, b)$	Uniform distribution 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

(i)

```
# Node Properties          |# CPDs of Nodes
numNodes: 4                |cpd(1):[1;0];
hidden: {1,3}              |cpd(2):(val(1),[N(50,0.05);
discrete: {1,3}            |                U(0,100)]);
node(1): ['Wo' 'Fa']       |
node(3): ['Wo' 'Fa']       |cpd(3):(val(1),[[0.99;0.01];
# Graph adjacency matrix   |                 [0.01;0.99]]);
graph: [0 1 1 0;           |
        0 0 0 1;           |cpd(4):(val(3),[N(val(2),0.05);
        0 0 0 1;           |                U(0,100)]);
        0 0 0 0]           |
```

(ii)

Fig. 3: (i) Conventions used in specifying the DPM; (ii) Configuration file for HMM-based view in Figure 1(i)

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 (discussed in Section V).

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).

B. Query Processing

Query processing over DPM-based views is simplified as a result of the particle-based representation in our system. A probabilistic query over a DPM-based view is executed by first converting it into a query over the corresponding particle table, 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 the query transformation process for the case of simple *select-project* queries over a single DPM-based view, and then consider more complex aggregate queries in Section IV-B.2. Designing robust query conversion algorithms for arbitrary queries is a topic of future work. 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.

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

bbqview(time, id, temp, humidity)

Here id denotes the sensor identifier and $temp$ is the temperature of the sensor. The unique *key* for this view is *(time, id)*,

Algorithm 1 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 *particles* 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 $> c\%$)

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 $Q'' \leftarrow$ *particles p2*
4: WHERE clause of $Q'' \leftarrow$ WHERE clause of Q
5: Add 'p1.key = p2.key' predicates to WHERE clause of Q'

and the schema of the corresponding particle table is:

particles(SampleID, time, id, temp, humidity, weight)

1) Select-Project SQL queries: Figure 4(i) shows a simple example of query transformation over this view. The query simply asks for the temperatures measured by all sensors. As we discussed in Section III, if the final result attributes in a query are probabilistic, the expected values are returned instead. The corresponding query over the particle table simply groups the particles by the key attributes and returns a weighted average of the temperature attribute.

The query in Figure 4(ii) 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 *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 ensures that only tuples with sufficient confidence are returned 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

```
SELECT id, time, temp FROM bbqview
                (a) Original Query
SELECT id, time, sum(temp*weight)
FROM particles GROUP BY id, time
                (b) Converted Query
```

(i)

```
SELECT id, temp FROM bbqview
WHERE temp > 20 AND time = 5
WITH CONFIDENCE 0.95
                (a) Original Query
SELECT id, sum(temp*weight)
FROM particles p1 WHERE time = 5
GROUP BY id HAVING 0.95 < (SELECT sum(weight)
      FROM particles p2 WHERE p1.id = p2.id
    AND  temp > 20 AND time = 5)
                (b) Converted Query
```

(ii)

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

(iii) Transformed AVG query (Original not shown)

```
WITH TBL1 AS (SELECT p.id AS pid, q.id AS qid, p.temp AS temp,
    p.weight AS weight, 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 prob
    FROM TBL1 GROUP BY pid, temp
    HAVING Count(*)= (SELECT COUNT distinct id FROM particles)+1
)
(a) VMinQ: SELECT sum(prob*weight*temp) FROM TBL2;
(b) EMinQ: SELECT pid, sum(prob*weight) FROM TBL2 GROUP BY pid;
```

(iv) Transformed Min Query (Original not shown)

Fig. 4: Examples of Query Transformation. Note that original queries are not shown for MIN and AVG.

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 III) 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; we omit the details due to space constraints.

The algorithms shown for query transformation assume that the key attributes are implicitly present in the query posed by the user. For instance, in Figure 4(i), the user explicitly requires *id* and *time* attributes, while in Figure 4(ii), the user explicitly requires the id attribute and implicitly the time attribute, by constraining it to be exactly 5. If such key attributes are not specified in the query, then additional processing needs to be performed for the above algorithms to perform query conversion correctly, in essence we need to include the key attributes in the converted query and only project them out in the final step of query execution.

2) Aggregate Queries: We now look at how to transform probabilistic aggregate queries. Prabhakar et al. [7] 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)

The query semantics we use for the aggregate queries is based on the possible worlds semantics.

Probabilistic AVG query (VAvgQ)
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 4(iii). However, 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 such 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 [7]. 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 4(iv).

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 measures 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.

C. Update Manager: Particle Filtering

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* [13] for this purpose. Particle filtering 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 III, this is exactly the internal representation that our system uses to maintain DPM-based views. Next we briefly describe the five routines of the particle filtering technique using the BBQ DPM (Figure 1(iii)). Pseudocodes for these routines and a more comprehensive illustration is presented in [17].

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 predicted using the state at time t. Specifically, for each existing particle at time t, a new particle for time $t+1$ is created by sampling from the relevant CPD. If (T_t^i, H_t^i) denotes the ith particle at time t, 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 $t+1$.

Filtering: The filtering procedure involves using the data that arrives at time $t+1$ to update the state estimate at time $t+1$. Each new particle is assigned a weight based on the values of the observed variables at time $t+1$. These weights are computed using the CPDs of the observed nodes. In our example, the weights are assigned to the predicted particles based on the CPD of the observed node M_t, $p(M_t|H_t)$. At the end of this step, the weights are normalized so they sum up to 1.

Re-sampling: Particle filtering may sometimes degenerate to the case where a single particle has all the weight. This is handled through a re-sampling step, where the current set of particles are re-sampled among themselves (based on weight) 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. 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 the model would attribute the reading to noise. Over time, as new readings arrive confirming the change, the inference process becomes more certain of the change in temperature. The earlier change that was attributed to noise, is now 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. This effect typically diminishes after a few steps, and we *backward update* the distribution of those steps that are atmost L time units away (where L is called the *smoothing lag*). The Smoothing step also reduces the variance of the filtering output. However, it is a very expensive operation - $O(N^2L)$ where N is the number of particles; and is hence not performed at every time step. This offers a trade-off between accuracy and performance wherein we can control the smoothing operation and its lag in order to meet user requirements.

V. Implementation Details

We built the prototype of our system using Java, and we use the open source Apache Derby (Java embedded database system) [2] to store 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 IV-C) 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 most important challenge we faced in our implementation was managing the many different possible types of node variables and their associated CPDs. Nodes may be continuous or discrete (with a variety of domains), may have any combination of discrete or continuous parent attributes and so on. To make the implementation generic and to provide the user with the flexibility to easily augment the system, we provide an extensible API that can be used to implement a new CPD:

- **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 (in **pVals**).
- **addSample(double val, ArrayList pVals):** This function adds a new data sample to the repository of samples used to learn this particular CPD.

- **computeParams():** This function, invoked after training samples are added, is used to "learn" parameters of the CPD.

VI. System Evaluation

In this section we present results from the experimental evaluation of our prototype implementation. Our experimental evaluation illustrates the need for using DPMs when dealing with erroneous and incomplete data streams, and demonstrates that our system is effective and efficient at applying DPMs to streaming data. Furthermore, our results also show that the mean squared errors obtained in the inference process follow the theoretically expected $1/N$ behavior [13].

A. Experimental setup

Dataset I: **Moving Objects Dataset**

Moving objects databases have received much attention in recent years [30], [32], [7]. We consider a moving objects scenario where a number of point objects with GPS devices constantly transmit their location to a central server. This data stream is assumed to be noisy and incomplete, and we would like to model it to infer the true locations and the velocities of the objects. Lacking a real-world dataset with GPS traces over multiple objects, we generate simulated data with the properties described above. We simulate a random linear trajectory for each object and add white Gaussian noise with a standard deviation of 2 units to the data. In addition, we randomly drop 5% of the readings to simulate incompleteness.

We use a KFM to infer the true locations and velocities (Figure 1(ii)). We enable the smoothing routine with a lag of 2. We model each moving object separately using a different KFM (different parameters), but store the information about all objects in a single table. 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 [11], [27], [7] 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 [21] that consists of traces from a 54-node sensor network deployment that measured light, humidity and temperature readings collected in a lab. The readings collected are extremely noisy and incomplete. Also, sensors that failed midway through deployment continued to transmit erroneous values. In our experiments, we attempt to accurately infer the temperature based on the observed humidity values. This is a common query processing strategy [11] in power-aware sensor networks, where acquiring all attributes is expensive. We run a series of processings tasks over this data.

Step 1: Remove Incorrect Data: Detect failure times of sensor nodes using an HMM-based view (Figure 1(i)) and remove all readings generated after this time.

Step 2: Learn DPM: Split the resulting data into training and testing datasets. Use training dataset (data collected for 6 days) to learn all CPDs of the DPM.

Step 3: Infer Temperature values: Use the humidity readings in the test dataset (data collected for 3 days) to infer the temperatures using the BBQ DPM (Figure 1(iii)).

Step 4: True Temperature Values: Determine exact temperature values by cleaning the observed temperatures using another DPM based view (not shown).

The resulting correct temperatures from Step 4 are compared with the temperatures inferred from Step 3 to evaluate the accuracy of the inferred temperatures.

B. Experimental Results

1. Applying DPMs to data is critical

Dataset I: The *intersection* query in Figure 5(i) measures the number of times at which two particles are closer than a specified distance δ. We execute this query on the raw GPS data and *kfview* and compare the number of correct intersections that are measured in both cases. Figure 5(ii) shows the plot comparing the percentage of missing intersections in the raw data and *kfview*. 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. *kfview* on the other hand, is able to capture most of the real intersections.

Dataset II: Figures 5(iii),(iv) show the results of executing Step 1, i.e., detecting the failure times for sensors. As we can see from Figure 5(iii), there are several incorrect values in the data after 500 hours (20 days approx), that need to be removed before we can use the data for learning. We also added a few simulated faults (iv) in order to further verify that the HMM-based view correctly identifies the faulty readings.

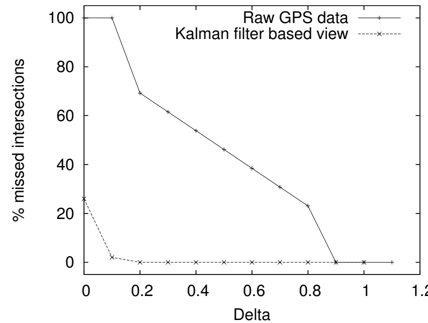
2. Inference using particle filtering is accurate

Dataset I: We execute the trajectory query shown in Figure 5(i), that returns the path traced by object 4, on the raw data and on *kfview*. 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 5(v). 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.)

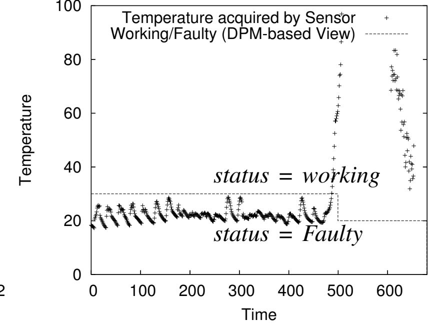
Dataset II: We compare the value of temperatures that were inferred in Step 3 (with just filtering, no smoothing) to the true temperature values generated in Step 4. We compute a mean square error estimate and plot the mean squared error as a function of the number of particles. We obtain the graph shown in Figure 5(v). 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. 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. We note here that queries over temperature (or other hidden variables) cannot be posed on the raw data as it was not explicitly measured. We can see that the error graphs for both datasets follow the theoretically estimated $(1/N)$ which validates our experiments.

```
(a) SELECT u.OID, v.OID, u.time
    FROM kfview u, kfview v
    WHERE (u.time = v.time)
    AND (|u.μ(x) - v.μ(x)| < δ)
    AND (|u.μ(y) - v.μ(y)| < δ)
(b) SELECT kfview.x, kfview.y
    FROM kfview
    WHERE kfview.OID = 4
```

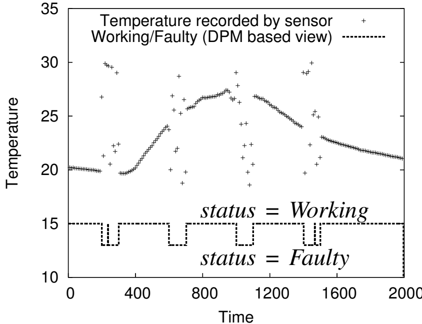
(i) (a) Intersection query (b) Trajectory query



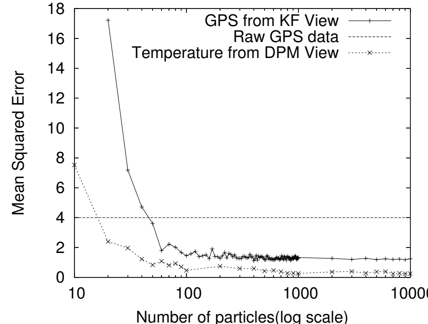
(ii) *kfview* captures all intersections



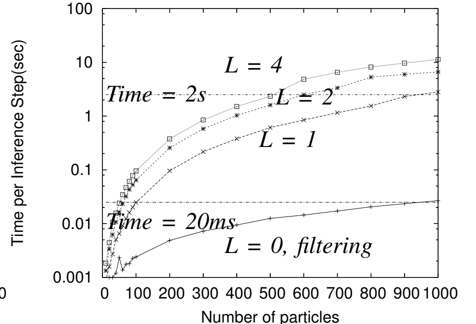
(iii) Faulty readings removed by HMM



(iv) Faulty readings removed by HMM view



(v) Accuracy of Inference



(vi) Smoothing performance

Fig. 5: (i) Queries used in Experiments; (ii) % of missed intersections as a function of δ on the raw data and the KFM-based view; (iii) Observed temperatures and the *working status* inferred using an HMM; (iv) Same as (iii) with simulated faults inserted; (v) Plot of mean-squared error vs number of particles for Dataset I and Dataset II. Mean squared error falls off as (1/N); (vi) Time taken for one inference step for various values of Smoothing Lag(L).

3. Inference using particle filtering is efficient

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 tuples(each of dimension 3) took 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. 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 5(vi). 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 (which means we can handle streams with 50 updates/second). 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 (still reasonable for most common streams). As the accuracy graph shows in Figure 5(v), 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.

VII. Related Work

Due to space constraints, we only discuss some of the most closely related work here.

Bayesian Networks and DPMs: Bayesian networks have been widely researched across a variety of research disciplines and numerous books have been written addressing their several aspects (e.g. [28], [16]). DPMs are a relatively newer and less well-established concept; they allow reasoning about temporal dimension as well, and are used extensively for modeling complex stochastic processes [26], [23]. Recently, DPMs have been seen as generalizations of well established concepts in seemingly disparate domains. 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 were observed fairly recently. Over the years, several general purpose toolkits have been developed that support Bayesian networks, and in some cases, dynamic Bayesian networks (e.g. [25]). 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 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 and incomplete. (e.g., [19], [3], [10],

[34], [31]). Several research efforts are underway to build systems to manage uncertain data (e.g. MYSTIQ [10], Trio [34], ORION [7], ConQuer [1]). As we discussed in Section III, views based on DPMs 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 [5], [6], [4], [24]. 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. There is also a large body of work on data collection from sensor networks that has applied higher-level techniques to sensor network data processing. Several systems propose to use probabilistic modeling techniques to answer queries over sensor networks [15], [11], [9], though these have typically used specific models rather than a generalized implementation in an existing relational database.

VIII. 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.

Acknowledgment

This work was supported by NSF Grants CNS-0509220 and IIS-0546136.

References

[1] Periklis Andritsos, Ariel Fuxman, and Renee J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.
[2] The Apache Derby Project. Web Site. `http://db.apache.org/derby/`.
[3] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE TKDE*, 4(5):487–502, 1992.
[4] D. Carney et al. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.
[5] Sirish Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.
[7] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.
[8] T. Choudhury, M. Philipose, D. Wyatt, and J. Lester. Towards activity databases: Using sensors and statistical models to summarize people's lives. *IEEE Data Eng. Bull.*, 29(1):49–58, 2006.
[9] 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.
[10] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.
[11] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.
[12] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84, 2006.
[13] Arnaud Doucet, Nando de Freitas, and Neil Gordon. *Sequential Monte Carlo methods in practice*. Springer, 2005.
[14] E. Hoke, J. Sun, and C. Faloutsos. Intemon: Intelligent system monitoring on large clusters. In *VLDB*, 2006.
[15] A. Jain, E. Change, and Y. Wang. Adaptive stream resource management using kalman filters. In *SIGMOD*, 2004.
[16] Michael I. Jordan. *Learning in Graphical Models (ed)*. MIT Press, 1998.
[17] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. Technical Report CS-TR-4867, Univ. of Maryland, 2007.
[18] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli. On-line fault detection of sensor measurements. *IEEE Sensors*, 2003.
[19] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *ACM TODS*, 22(3), 1997.
[20] D. Lymberopoulos, A.S. Ogale, A. Savvides, and Y. Aloimonos. A sensory grammar for inferring behaviors in sensor networks. In *IPSN*, 2006.
[21] Sam Madden. Intel lab data, 2004. `http://berkeley.intel-research.net/labdata`.
[22] A. M. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, 2002.
[23] V. Mihajlovic and M. Petkovic. Dynamic bayesian networks: A state of the art. University of Twente Document Repository 2001.
[24] Rajeev Motwani et al. Query processing, resource management, and approximation in a data stream management system. In *CIDR*, 2003.
[25] Kevin Murphy. The Bayes net toolbox for matlab. *Computing Science and Statistics*, 33, 2001.
[26] Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learnig*. PhD thesis, UC Berkeley, 2002.
[27] D. Patterson, L. Liao, D. Fox, and H. Kautz. Inferring high level behavior from low level sensors. In *UBICOMP*, 2003.
[28] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
[29] L Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. 77:257–286, 1989.
[30] K. Raptopoulou, M. Vassilakopoulos, and Y. Manolopoulos. Towards quadtree-based moving objects databases. *Lecture Notes in Computer Science, Volume 3255, Jan 2004*, 2004.
[31] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.
[32] G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. Managing uncertainty in moving objects databases. *ACM TODS*, 29(3), 2004.
[33] G. Welch and G. Bishop. An introduction to the Kalman filter. `http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html`.
[34] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.
[35] Jie Ying et al. A hidden markov model-based algorithm for fault diagnosis with partial and imperfect tests. *IEEE Trans. on Systems, Man, and Cybernetics, Part C*, 2000.