

## Abstract

Title of Dissertation: SCALABLE STATISTICAL MODELING AND QUERY PROCESSING OVER LARGE SCALE UNCERTAIN DATABASES

Bhargav Kanagal Shamanna  
Doctor of Philosophy, 2011

Dissertation directed by: Dr. Amol Deshpande  
Dept. of Computer Science

The past decade has witnessed a large number of novel applications that generate imprecise, uncertain and incomplete data. Examples include monitoring infrastructures like RFIDs, sensor networks and web-based applications such as information extraction, data integration etc. In my dissertation, I addressed the challenges in managing uncertain data and developed algorithms for efficiently executing queries over large volumes of such data.

First, for meaningful analysis of such data, we need the ability to remove noise and infer useful information from uncertain data. To address this challenge, I developed a declarative system for applying *probabilistic models* to databases and *data streams*. The output of such probabilistic modeling is *probabilistic data*, i.e., data annotated with probabilities of correctness. Often, the data exhibits strong correlations. To manage such data, I built a probabilistic database system that can manage large-scale correlations and developed algorithms for efficient query evaluation.

My system allows users to provide uncertain data as input and to specify arbitrary correlations among the entries in the database. In the back end, we represent correlations as a forest of “junction trees”, an alternative representation for probabilistic graphical models (PGM). We execute queries over the probabilistic database by transforming them into message passing algorithms (inference) over the junction tree. Traditional algorithms over junction trees typically require accessing the entire tree, making them infeasible for large-scale correlated databases. Hence, I developed an index data structure over the junction tree called INDSEP that allows us to circumvent this process and scalably evaluate inference queries, aggregation queries and SQL queries over the probabilistic database.

Next, I augmented the existing query evaluation model so that users can better understand query results. The existing query evaluation model returns output tuples along with their probability values, which provides very little intuition to the users: for instance, a user might want to know “Why does this output tuple have such high probability?” or “Which are the most influential input tuples for my query ?” Hence, I developed algorithms to compute *explanations* for query results and perform *sensitivity analysis* for users to better understand query results.

SCALABLE STATISTICAL MODELING AND QUERY  
PROCESSING OVER LARGE SCALE UNCERTAIN  
DATABASES

by

Bhargav Kanagal Shamanna

Dissertation submitted to the Faculty of the Graduate School  
of the University of Maryland, College Park in partial  
fulfillment of the requirements for the degree of  
Doctor of Philosophy  
2011

Advisory Committee:  
Professor Amol Deshpande, Chair  
Professor Lise Getoor  
Professor David Jacobs  
Professor Hector Bravo  
Professor Mark Austin

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivating Applications . . . . .	3
1.1.1	Event Monitoring & Complex Event Processing . . . . .	3
1.1.2	Information Extraction/Integration System . . . . .	7
1.1.3	Probabilistic Modeling of Data Streams [60] . . . . .	13
1.2	Problem/Research Challenges . . . . .	16
1.2.1	Probabilistic Modeling of Uncertain Data . . . . .	16
1.2.2	Query Processing over Correlated Probabilistic Data . . . . .	17
1.2.2.1	Challenges in dealing with Correlations . . . . .	20
1.2.3	Robust query processing: Sensitivity & Explanations . . . . .	22
1.3	Our Approach . . . . .	23
1.4	Outline & Research Contributions . . . . .	27
<b>2</b>	<b>Background and Related Work</b>	<b>33</b>
2.1	Background . . . . .	33
2.1.1	Probabilistic Modeling of Uncertain Data . . . . .	35
2.1.1.1	Hidden Markov models (HMMs) . . . . .	35
2.1.1.2	Linear Dynamical Systems . . . . .	37
2.1.1.3	DPMs: Graphical Representation . . . . .	38
2.1.1.4	Inference in DPMs . . . . .	40
2.1.2	PGM Representation . . . . .	40
2.1.3	Query Processing over PGMs . . . . .	42
2.1.4	Junction Tree Representation of PGMs . . . . .	46
2.1.5	Query Processing over Junction Trees . . . . .	50
2.1.6	Special Case 1: Markovian streams . . . . .	56
2.1.7	Special Case 2: Tuple Independent Probabilistic Databases . . . . .	56
2.1.7.1	Queries . . . . .	57
2.1.7.2	Detecting read-once lineages . . . . .	59
2.1.7.3	Shannon Expansions . . . . .	61
2.2	Related Work . . . . .	62
2.2.1	Probabilistic Databases . . . . .	62
2.2.2	Inference in Graphical Models . . . . .	65

2.2.3	Indexes for Probabilistic Databases . . . . .	67
2.2.4	Sensitivity Analysis . . . . .	68
<b>3</b>	<b>PrDB System Overview</b>	<b>70</b>
3.1	Relational Storage System . . . . .	72
3.2	Parser and Language . . . . .	74
3.2.1	Parser Implementation . . . . .	77
3.2.2	Factor Semantics . . . . .	78
3.3	Junction tree & INDSEP . . . . .	79
3.4	Query Processor . . . . .	79
3.5	Probabilistic Modeling System . . . . .	81
3.5.1	Specifying DPM-based Views . . . . .	82
<b>4</b>	<b>Probabilistic Modeling of Uncertain Data</b>	<b>86</b>
4.1	DPMs as Database Views . . . . .	87
4.2	Design . . . . .	91
4.3	Update Manager: Particle Filtering . . . . .	91
4.4	System Evaluation . . . . .	93
4.4.1	Experimental setup . . . . .	94
4.4.2	Experimental Results . . . . .	97
<b>5</b>	<b>INDSEP</b>	<b>100</b>
5.1	INDSEP Data Structure . . . . .	101
5.1.1	Overview of the INDSEP Structure . . . . .	102
5.1.2	Shortcut Potentials . . . . .	104
5.2	Index Construction . . . . .	107
5.2.1	Hierarchical Partitioning . . . . .	107
5.2.2	Variable Renaming . . . . .	109
5.2.3	Assigning Range Lists and Add Lists . . . . .	110
5.3	Query Processing . . . . .	110
5.3.1	Inference/Extraction Queries . . . . .	110
5.3.2	Aggregate Queries . . . . .	113
5.4	Handling Updates . . . . .	115
5.4.1	Updates to Existing Potentials . . . . .	116
5.4.2	Inserting New Data . . . . .	119
5.4.3	Deletions . . . . .	121
5.5	Experimental Evaluation . . . . .	121
5.5.1	Implementation Details . . . . .	121
5.5.2	Experimental Setup . . . . .	122
5.5.3	Results . . . . .	125

<b>6</b>	<b>Lineage Processing over INDSEP</b>	<b>130</b>
6.1	Lineage Processing Algorithms over Junction trees . . . . .	131
6.1.1	Message Passing for Lineage Processing . . . . .	132
6.1.2	Pivot Selection . . . . .	137
6.1.3	Dealing with Disconnections . . . . .	138
6.2	Lineage Processing using INDSEP . . . . .	139
6.2.1	Recursive Approach . . . . .	140
6.2.2	Shortcomings . . . . .	143
6.3	Lineage Planning & Evaluation . . . . .	145
6.3.1	Lineage Planning . . . . .	146
6.3.2	Lineage Plan and Execution . . . . .	149
6.3.3	Approximation Technique . . . . .	151
6.4	Experimental Evaluation . . . . .	152
6.4.1	Implementation Details . . . . .	152
6.4.2	Experimental Setup . . . . .	152
<b>7</b>	<b>Query Processing on Markovian Sequences</b>	<b>159</b>
7.1	Markovian Sequences . . . . .	160
7.2	Probabilistic Sequence Algebra . . . . .	162
7.3	Operator Algorithms . . . . .	165
7.4	Query Evaluation . . . . .	174
7.4.1	Query Syntax . . . . .	174
7.4.2	Query Planning and Optimization . . . . .	175
7.4.3	Approximation Strategies . . . . .	178
7.5	Experiments . . . . .	178
7.5.1	Experimental Setup . . . . .	179
7.5.2	Experimental Results . . . . .	180
<b>8</b>	<b>Robust Query Processing for Probabilistic Databases</b>	<b>187</b>
8.1	Formal Problem Statement . . . . .	188
8.1.1	Sensitivity Analysis . . . . .	188
8.1.2	Explanation Analysis . . . . .	190
8.1.3	Warmup: SUM/COUNT . . . . .	191
8.1.4	Relation to Meliou et al. [77] . . . . .	191
8.2	Sensitivity Analysis . . . . .	192
8.2.1	Value queries . . . . .	192
8.2.1.1	Boolean conjunctive queries . . . . .	193
8.2.1.2	Conjunctive queries . . . . .	198
8.2.1.3	Aggregation queries . . . . .	199
8.2.2	Set queries . . . . .	201
8.2.2.1	Probabilistic Threshold Queries . . . . .	201
8.2.2.2	Top-k queries by probability . . . . .	204

8.2.3	How is $\epsilon$ assigned? . . . . .	204
8.3	Explanation Analysis . . . . .	205
8.3.1	Boolean Conjunctive Queries . . . . .	205
8.3.2	Aggregation queries . . . . .	208
8.4	Incremental Recomputation . . . . .	209
8.4.1	Conjunctive Queries . . . . .	210
8.4.2	Aggregation . . . . .	212
8.5	Experimental Evaluation . . . . .	215
8.5.1	Experimental Results . . . . .	216
<b>9</b>	<b>Conclusions</b>	<b>220</b>

# List of Figures

1.1	RFID Event Monitoring Application . . . . .	4
1.2	Data extracted by an information extraction engine. The CarAds table in part(a) is extracted by the engine. It has both tuple uncertainty and attribute uncertainty. In part(b), we show an equivalent representation of the CarAds table by converting the attribute uncertainty to tuple uncertainty. The correlations are indicated in part (c). . . . .	9
2.1	Graphical representations of DPMs. (i) Using an HMM for fault detection; (ii) Using a KFM for velocity and location estimation. (iii) Parameters of the HMM model. (iv) Parameters of the KFM model. . . . .	36
2.2	Example: (a) A probabilistic database $D^p$ on two relations $R_1$ and $R_2$ exhibiting both tuple-existence and attribute-value uncertainties (e.g. $a$ indicates the random variable corresponding to the existence of the first tuple in $R_1$ ); (b) The directed PGM that captures the correlations in $D^p$ , and (c) the junction tree representation of the PGM. . . . .	41
2.3	Query processing over probabilistic databases using graphical models: (a) a graphical model over 4 attributes; (b) an example set of CPDs for the graphical model (bold-faced variables indicate the child nodes); (c, d) to execute a query over the probabilistic database, we add new variables to the PGM and introduce additional CPDs; (e) PGM for evaluating the lineage query (f) AND factor for variable $E$ ; (g) OR factor for variable $O$	43



2.4	Figure shows the various steps in the construction of the junction tree from the original PGM representation in part(a). In part(b) we moralize the PGM by “marrying” (connecting) nodes $C$ and $D$ . In part(c) we triangulate the resulting graph by adding an edge between nodes $C$ and $B$ . Note that an edge between $A$ and $D$ would also work. In part(d), we construct the clique graph. The weight of each edge is indicated and the maximum spanning tree is also shown. The final junction tree is shown in part(e). . . . .	47
2.5	Path constructed for query $\{e,o\}$ . . . . .	50
2.6	Figure shows (i) a tuple uncertain probabilistic database (ii) the graphical model that captures the correlations among the various tuples . . . . .	54
2.7	Example of a Markov sequence . . . . .	56
2.8	(i) Boolean formula $(x_1x_2+x_3x_4)(x_5x_6+x_7)$ represented using an AND/OR tree. Leaves denote variables of the formula, internal nodes are intermediate expressions. (ii, iii, iv) Steps involved in generating the read-once formula for $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4) \vee (x_4 \wedge x_1)$ . . . . .	57
3.1	Various components of the PrDB system. Probabilistic data is inserted by the user via the parser. The correlations are stored in the factor tables in the system. We propose a junction tree-based materialized view representation of the database and index it using the <i>INDSEP</i> data structure. The query processor interacts with the <i>INDSEP</i> and the underlying user tables. . .	71
3.2	Schema of the PrDB model. User tables are shown in (a). Internal system tables are shown in (b) . . . . .	75
3.3	Various constructs to insert data and correlations in PrDB . .	76
3.4	List of statements to generate the database of Figure 3.2 . . .	77
3.5	(i) Conventions used in specifying the DPM; (ii) Configuration file for HMM-based view in Figure 2.1(i) . . . . .	83
4.1	(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) . . . . .	88
4.2	Graphical representation of the BBQ DPM used for modeling Intel Lab data (Section 6.4) . . . . .	88

4.3	(i) Queries used in Experiments; (ii) % of missed intersections as a function of $\delta$ on the raw data and the KFM-based view; (iii) Observed temperatures and the <i>working status</i> 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). . . . .	95
5.1	(a) shows a hierarchical partition of the junction tree shown in Figure 2.2(c). Note that the separator nodes separating two partitions are replicated in both the partitions. The corresponding <i>INDSEP</i> data structure is shown in (b).The contents of the index node $I_2$ is shown in part(c). . . . .	101
5.2	Illustrating overlays and shortcut potentials. Using the cached potential $p(C, D)$ allows us to shortcut the clique $C_2$ completely.	106
5.3	The Steiner trees generated at different index nodes while executing the inference query $\{e,o\}$ on the junction tree in Figure 2.2(c) is shown in (a), (b), (c). (d) shows the final graphical model generated as a result of the extraction query $\{e,o\}$ . The junction tree generated by the aggregation query is shown in (e)	116
5.4	We generate the Markovian sequence database using the schema shown in (a). The junction tree structure of the Markovian sequence is shown in (b) . . . . .	122
5.5	Illustrating query performance in terms of number of blocks accessed and cpu time for workloads $W_1, W_2, W_3$ and $W_4$ when index data structure is absent, index is present without shortcut potential, both index and shortcut potential are present. (a) & (b) correspond to the event database, while (c) & (d) correspond to the Markovian sequence database. We note that the graph is in <i>logarithmic scale</i> , so the gains are substantially more than what is apparent. . . . .	124
5.6	As shown in part (a), the time taken to bulk load the index is linear in the size of the database. Part (b) shows that the height of the tree increases in a logarithmic-like fashion as the size of the database increases. Part (c) shows that as the disk block size increases, the amount of approximation reduces, i.e., less than 20% for 4kB block size. . . . .	125

5.7	Graph in part (a) shows that the query performance (as function of number of blocks) improves when block size is increased. The bar graphs in parts (b) & (c) show that the query performance falls as the percentage of shortcut potentials that are out-of-date increases. Part (d) shows the update times – as we can see, the overheads for updating INDSEP data structure are quite minimal. . . . .	127
6.1	Different stages in the simplification process . . . . .	132
6.2	Figure shows (i) a tuple uncertain probabilistic database (ii) the graphical model that captures the correlations among the various tuples (iii) its equivalent junction tree (iv) the INDSEP data structure corresponding to the junction tree in (iii) and (v) Steiner tree computed while evaluating the inference query $\{a, k\}$ . Pivot clique (ac) is shaded. (vi) The childTree stored in the root is shown here. Note that $I_2$ is connected to $I_1$ via $c$ and to $I_3$ via $j$ as indicated in (iii). Note that (i),(ii) and (iii) are repeated from Chapter 2 for convenience. . . . .	133
6.3	(a) Illustrating the order of multiplication and simplification in Eager+Order heuristic. Initially, we multiply pdfs $p(f, h)$ and $p(c, f, g)$ since that edge has the maximum weight. (b) When pivot = (cfg), the sequence of messages passed is indicated above the graph (right arrows). When pivot = (ab), the sequence of messages is indicated below the graph (left arrows). . . . .	138
6.4	(i) illustrates the computation of subexpressions (ii) shows an intermediate junction tree generated in the root node while processing $((d \vee e) \wedge (n \vee o)) \vee (b \wedge c)$ . . . . .	140
6.5	PrDB's lineage processing component overview: Input conjunctive query is first executed by the relational engine which computes lineages of output tuples. Lineage Planner then computes an optimal plan for processing these lineages, which is executed by the Lineage Processor. . . . .	145
6.6	Lineage Plan for lineage $\lambda = (d \vee e)(n \vee o) \vee (b \wedge c)$ . . . . .	149
6.7	Results: (a) Processing lineages using INDSEP is more scalable. (b) and (c) Illustrating benefits of EAGER+ORDER heuristic over the EAGER and naive approaches for datasets $D_2$ and $D_1$ . (d) As the correlations increase, lineage processing times increase. Special purpose technique (SPT) performs better for the independent dataset $D_1$ . (e) and (f) Illustrating benefit of batch lineage processing. (g) Sampling errors can be reduced by increasing number of samples. (h) INDSEP improves the quality of our approximations significantly. . . . .	158

7.1	(a) Example of a Markovian sequence $S^p$ on attributes $X$ and $Y$ ; (b) Schema graph and (c) clique list of $S^p$ ; (d) Representing $S^p$ using one relation per CPD. . . . .	162
7.2	(a) Executing a selection predicate ( $X > Y$ ) entails adding new <i>exists</i> variables ( $E^i$ ); the dotted edges show the changes to the schema. (b) Projection may result in a non-Markovian sequence – if $Y$ nodes are eliminated, the resulting $X$ sequence (shown through dotted edges) is not Markov. . . . .	167
7.3	Constructing CPDs for new nodes for (a) selection, (b) aggregate, and (c) aggregate with selection ( $dom(X) = dom(Y) = \{0, 1\}$ ). . . . .	168
7.4	Illustrating aggregate computation. $G^i = Agg(X^1, X^2, \dots, X^i)$ . In each <code>get_next()</code> call, dotted variables are added to the PGM, and the boxed nodes are eliminated, continuously maintaining state $p(X^i, G^i)$ and $p(X^i, G^i, E^i)$ respectively. Also, note the dependence of $G^i$ on $E^i$ in (b). . . . .	170
7.5	(a) PGM for sliding window aggregate. $G^i$ 's denote the aggregates that we have to compute. (b) After eliminating the $X^i$ variables, we obtain a clique on the $G^i$ variables, which is #P-hard. (c) Hence, we split the PGM into components as shown. Unmarked nodes are intermediate aggregates. (d) For tumbling window aggregates, we only eliminate boxed nodes to obtain the Markovian sequence shown in (e). Removing nodes $X^3$ and $X^6$ is postponed to a later projection. . . . .	173
7.6	The set of queries and the schemas used in the experiments. . . . .	180
7.7	(a) We plot the % error in query processing for various operators when temporal correlations are ignored, (b) We show performance (throughput) of the windowing operators, (c) We show the throughput of aggregate operators for different cases. . . . .	182
7.8	Figures (a),(b),(c),(d) illustrate query optimization. (a),(b) show the need for determining the correct location for the projection operator. (c) demonstrates gains made by deleting redundant edges in the model. Note that this is not drawn to scale, only used for comparison. (e),(f) demonstrate advantages of our system over previous approaches. Notice that the y-axis is in the log scale, so our gains are substantial. (g),(h) describe accuracy and performance for the approximate map operator . . . . .	183

8.1 Results: (a) Top-k queries by probability are sensitive to input probabilities. As we modify the probability  $p(x)$ , the output probabilities and the top-k output change as shown. (b,c) Here we demonstrate that sensitivity analysis can be implemented very efficiently. The overhead above computing output probabilities for TPC-H queries is at most 5%. (d) The break up of the times spent in the different components of the algorithm. S.A refers to sensitivity analysis. (e) Same as part(d) for Boolean TPC-H queries (f) Illustration of the benefits of pruning algorithms. (g) Explanation analysis is efficient. (h) Incremental recomputation for boolean conjunctive queries is efficient. . . 219

# Chapter 1

## Introduction

As we continue to advance in the Information Age, massive amounts of data are being generated at a rapidly increasing rate by a large number of applications. While database systems were developed to successfully manage large amounts of data and query them efficiently, they are incapable of dealing with the array of new types of data being generated by new and emerging applications. The past decade has witnessed various such types of data generated in arenas like *sensor networks* and other distributed monitoring infrastructures, which collect measurements using tiny low-cost sensors [2], and in emerging applications like *information extraction* and *data integration* which gather data by crawling the countless numbers of web sites on the web [36, 54, 46]. Other data sources include Bioinformatics [38], social networks [1], mobile data sources [18, 104, 57], images and video data [84, 85, 83] and so on. Apart from the extensive scale at which these applications continue to deliver data, these applications also share a very important property: much of the data generated in these applications is *imprecise* and *uncertain*. The imprecision in the data is due to a variety of reasons. For instance, the push for mass produc-

ing very low cost sensing devices has led to them being inaccurate and failure prone – this leads not only to inaccurate measurements, but also incomplete measurements when the sensors fail. Extracting structured information from web data involves natural language processing [36, 48], which is error prone – i.e., we might either extract wrong information from natural language text or fail to extract correct information. Similarly, in Bioinformatics, ambiguities in computation of genome sequences arise due to the experimental noise and the sheer complexity of DNA sequencing experiments [47]. Finally, the current state of the art in computer vision [41] does not allow us to precisely recognize objects and extract corresponding events from images and video, thereby resulting in uncertain data.

While traditional database systems can manage *deterministic* data efficiently and provide declarative methods for querying them using SQL, they currently cannot support uncertain data, even more so when the data is *correlated*. However, the end-users of the applications mentioned above still want to be able to analyze such data and obtain interesting and novel information from them: for instance, the scientists in a wireless sensor networking application would be interested to know whether a remote sensor is failing (perhaps by looking at the signature of the measurements made by the sensor) or in an RFID-based inventory management system, the store owners might be interested to know if a product (which is associated with an RFID tag) has been successfully checked out of the store, or in other words, the likelihood of a theft. Hence, there is a need for developing novel database systems that can efficiently manage large-scale uncertain data and provide query processing support. However, building such a system can be quite challenging. We illustrate this with a few motivating applications next. We start by describing a

real world application that requires us to reason about correlated probabilistic data in an event processing application.

## 1.1 Motivating Applications

### 1.1.1 Event Monitoring & Complex Event Processing

Consider an RFID-based event monitoring application [91, 71, 103] that uses RFID tags and readers in order to monitor various entities such as personnel and hardware in a building. A building is instrumented with RFID devices that detects RFID tags in its vicinity. Using these readings, the application detects the occurrence of different types of events in the building. Raw RFID data is typically noisy and incomplete. RFID tags sometimes go undetected by the tag readers, or sometimes they are detected multiple times. Therefore, it is subjected to *probabilistic modeling* to remove such noise and also to infer the missing values from the data. This application is schematically indicated in Figure 1.1. As shown in the figure, raw RFID data is subjected to probabilistic modeling. For instance, if we detect the presence of a computer *PC* at RFID device *X* and subsequently at *Z*, then clearly we must have also detected the PC either at  $Y_1$  or  $Y_2$  at an intermediate time - this information may be missing however in the raw RFID readings, due to noise.

The output of the above probabilistic modeling is a set of uncertain events associated with *occurrence probabilities*. For instance, the event `entered(Mary, conf-room, 2:10pm)`, which says that Mary was found near the conference room at 3:10pm, may be assigned a probability of 0.6 of actually having occurred (Figure 1.1). Similarly, the event `coffee(Bob, 2:05pm)`, which indi-



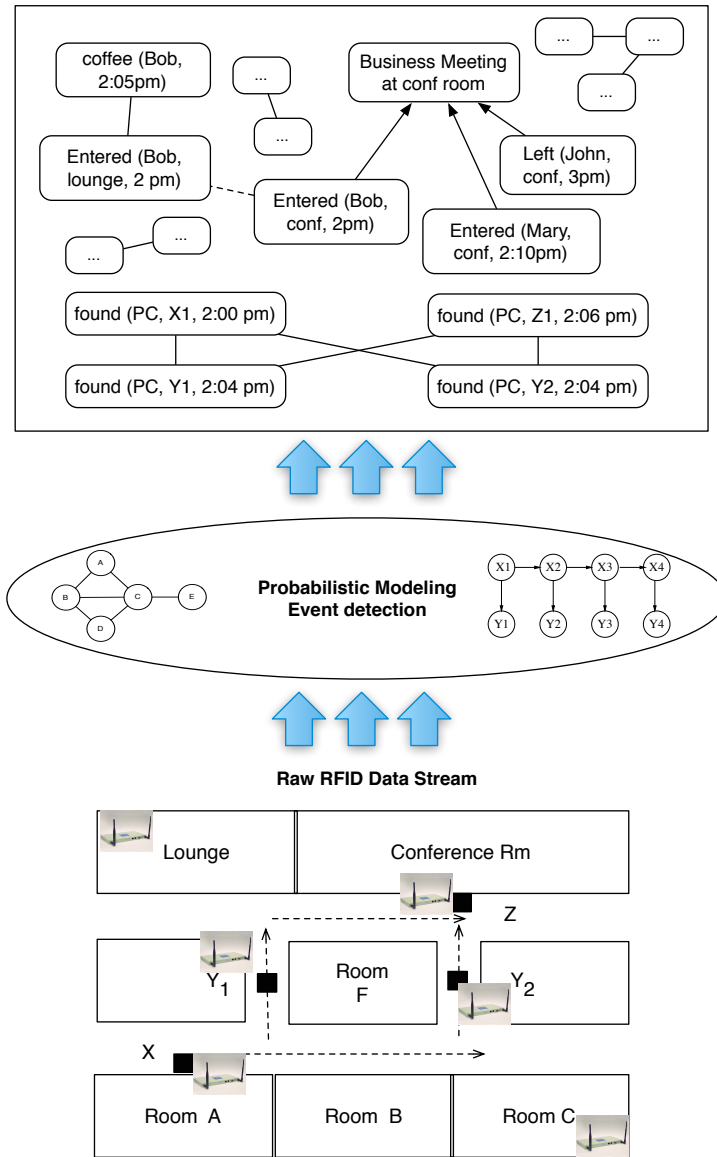


Figure 1.1: RFID Event Monitoring Application

cates that Bob was making coffee at 2:10pm, may be assigned another probability. These uncertain events are naturally *highly correlated* with each other. For example, the event `coffee(Bob, 2:05pm)` is strongly *positively* correlated with the event `entered(Bob, lounge, 2:00pm)` since the coffee machine is present in the lounge. However, the events `entered(Bob, lounge, 2pm)` and

`entered(Bob, conf-room, 2pm)` must be *mutually exclusive* since Bob cannot be found at the same time in two different locations. This situation occurred due to inaccurate detection by the RFID device. Additional correlations arise when *compound events* are *inferred* from basic events. For instance, the occurrence of the event `business-meeting(conf-room)` is directly dependent on the events `entered(Mary, conf-room, 2:10pm)` and `entered(Bob, conf-room, 2:00pm)`. Such correlations are typically indicated by drawing a graph over the events and adding (possibly directed) edges between correlated events as shown in Figure 1.1. The nature of the correlation itself can be quantified by associating probability distributions or constraints over the corresponding events.

Given such a large collection of uncertain data, the monitoring applications and the users may ask a wide variety of queries.

1. **(Inference)** “What is the likelihood that Bob and Mary attended a meeting given that John did not attend?”. This is an example of a *inference* query. As has been observed in much work before [96, 61, 71, 66], ignoring the correlations can result in highly inaccurate results to such queries.
2. **(Aggregation)** “How many business meetings occurred over the last week”. This is an example of an *aggregation* query. Since there is a probability of occurrence for each business meeting, we need to compute and return a probability distribution on the aggregate value. Although the expected value of the aggregate can be computed efficiently, computing a *probability distribution* over the result requires reasoning about the correlations.

3. **(Lineage/Boolean Formula)** Other queries in such a system involve computing the probabilities of *compound events* expressed as compositions of simple events. For instance, a query of interest is: *What is the probability that the PC was transferred correctly from its starting location at Room A to the final location at the conference room?* Since there are three RFID devices  $X_1$ ,  $X_2$  and  $X_3$  between Room A and the conference room, the compound event corresponding to the successful transfer of the PC is given by the boolean conjunction:

$$\text{obs}(X_1, \text{'PC'}, 10:00) \wedge \text{obs}(X_2, \text{'PC'}, 10:05) \wedge \text{obs}(X_3, \text{'PC'}, 10:10)$$

We now need to compute the probability of the above boolean formula. We call such a query, where we have to compute the probability of a boolean formula, a *lineage* query.

After evaluating the query, the user is provided with the output probabilities, for example, the likelihood that the PC was correctly transferred is 0.2. On receiving the output, the user may have additional concerns. For instance, the user may have the following concerns:

1. **(Explanations)** Why is the probability of having correctly transferred the above PC so low ? Here, the user is interested to know the cause/explanation for the output tuple. Note that due to large-scale correlations, the explanations for a certain output tuple may involve input tuples that are not explicitly part of the boolean formula.
2. **(Sensitivity Analysis)** What are the most sensitive input tuples for the output probability. In this case, the user is interested to know the most sensitive input tuples that can alter this probability. The reason for this

is because the user may be unsure about certain input probabilities and would be interested to know which tuples can influence the probability the most. Just as with explanations, due to the presence of large-scale correlations, the sensitive input tuples need not be the tuples forming the boolean formula.

3. **(Re-evaluation)** How will the output probabilities change if I alter some of the input probabilities, e.g., by resolving the uncertainty of `obs(X2, 'PC', 10:04)`. After determining the sensitive input tuples and choosing to resolve them (e.g., by querying an expert), the user would be interested to re-evaluate the query and obtain results quickly.

### 1.1.2 Information Extraction/Integration System

Consider an Information extraction/integration system [46, 54, 78, 36] that scans used car advertisements from multiple different sources like `cars.com`, `craigslist.com`, and `autotrader.com` and populates a relational database with structured data, shown in Figure 1.2. To cope with the enormous amounts of data on the web, the system may employ automatic extractors to detect potential tuples. Since most web data is in natural language format, it is impossible to extract only the correct tuples: there may be tuples that do not necessarily belong to the database. Also, tuples may be extracted from web sites that may be out of date. In order to handle such uncertainty, the IE system may choose to assign weights to the extracted tuples that denote the *degree of correctness* or validity of the tuple. These weights are interpreted as *probabilities* of existence of the tuples. The information extraction system uses entity recognition (e.g., identifying people, locations, companies), rela-

tionship detection (e.g., affiliations), sentiment analysis (e.g., from reviews), entity resolution (e.g., “Y! Labs” and “Yahoo! Labs” refer to the same entity), and other complex machine learning algorithm to compute probabilities. These probabilities are shown in Figure 1.2(a) under the column labeled ‘?’. The probabilities can be obtained via machine learning algorithms based on CRFs [46] or Bayesian inference techniques [54]. Larger probability values indicate that the tuple is extracted correctly and has a better chance of belonging to the table, while smaller probabilities indicate that the tuple has a lower chance of belonging to the table. Uncertainties can also occur in the values of the attributes. For instance, for tuple  $x_3$  in Figure 1.2(a), the value of the attribute `Model` is unknown. It can either be ‘Honda’ with probability 0.5 or ‘Toyota’ with probability 0.4 or ‘Ford’ with probability 0.1. This might have happened either due to the fact that the model was missing in the advertisement, or the information extraction system could not extract the information corresponding to the model of the car. Figure 1.2(a,c,d) shows three relations extracted automatically from the web, along with the probability attached to each tuple.

In addition, the tuples in such a database may exhibit significant correlations; e.g., both tuples  $x_1$  and  $x_2$  in Figure 1.2(a) cannot belong to the relation simultaneously, since they correspond to the same car (since the VIN is same), but their prices/sellers are different. This situation occurred since we gleaned the data from different sources with conflicting information, only one of which is really correct. This corresponds to a *mutual exclusion* correlation between the two tuples, where the presence of one tuple necessarily implies the absence of the other tuple. We may denote such correlations using a graph (Figure 1.2(e)). Note that  $x_1$  is connected to  $x_2$  indicating that they are cor-

<u>tid</u>	VIN	Seller	Model	Price	?
$x_1$	1A0	239	Honda	3500	0.3
$x_2$	1A0	231	Honda	4500	0.8
$x_3$	2B1	231	{Honda 0.5, Toyota 0.4, Ford 0.1}	4500	1

(a) CarAds (tuple/attribute uncertainty)

<u>tid</u>	VIN	Seller	Model	Price	?
$x_1$	1A0	239	Honda	3500	0.3
$x_2$	1A0	231	Honda	4500	0.8
$x_3$	2B1	231	Honda	4500	0.5
$x_4$	2B1	231	Toyota	4500	0.4
$x_5$	2B1	231	Ford	4500	0.1

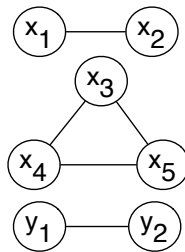
(b) CarAds (only tuple uncertainty)

<u>lid</u>	Seller	Name	Address	?
$y_1$	239	Honda of College Park	12345	0.3
$y_2$	239	Honda Company, College Park	12345	0.3
$y_3$	231	Ford Company	12207	0.8
$y_4$	340	Toyota Car Company	12209	0.9

(c) Location

<u>rid</u>	Seller	Reputed	?
$z_1$	239	Good	0.3
$z_2$	231	Bad	0.7
$z_3$	340	Good	0.9

(d) Reputation



(e) Correlations

239	$x_1(y_1 + y_2)$	0.8
-----	------------------	-----

(f) Result for query  $Q_1$ 

Figure 1.2: Data extracted by an information extraction engine. The CarAds table in part(a) is extracted by the engine. It has both tuple uncertainty and attribute uncertainty. In part(b), we show an equivalent representation of the CarAds table by converting the attribute uncertainty to tuple uncertainty. The correlations are indicated in part (c).

related. Additional correlations occur due to *attribute uncertainty*. A possible approach to handle attribute uncertainty is by converting it into tuple uncertainty: if the attribute ‘Model’ was missing for the extracted tuple, the system

adds separate tuples for each possible Model offered by the seller, along with a mutual exclusion dependency among the alternatives. For example, suppose that the tuple  $x_3$  in Figure 1.2(a) has three possible alternatives for the Model attribute. In this case, the system adds three tuples  $x_3, x_4, x_5$  as shown in Figure 1.2(b), along with a mutual exclusion dependency among them ( $x_3, x_4$  and  $x_5$  in Figure 1.2(e)). In general, complex correlations can arise in any application that uses sophisticated learning and inference techniques like Bayesian networks [60, 91, 71].

Given such a database, the users can pose several interesting queries. We provide examples of queries below. As before, we include *aggregation*, *lineage* and *inference* queries.

1. **(Aggregation)** “How many Honda cars are available for sale ?” This is an aggregation query that requires us to compute the probability distribution of the number of Honda cars available for sale. While computing an expected value of the output is easy via *linearity of expectation*, computing the entire probability distribution is non-trivial in the presence of correlations.
2. **(Lineage)** “Give me the list of all reputed auto dealerships near the zip code 12345”. In a deterministic database, this corresponds to executing the relational algebra query, expressed in SQL:

```
SELECT Location.Seller FROM Location, Reputation
WHERE Location.Address = '12345'
```

This basically corresponds to a join between the relations *Location* and *Reputation* preceded by the selection of appropriate zip code (12345) in *Location* and the appropriate reputation. This is commonly known

as a *conjunctive* query in database query evaluation. However, how to process the query is not immediately clear in our example since tuples do not exist with certainty, i.e., they need not belong to the relation. Consequently, some of the tuples need not participate in the join described above, leading to different possible results in different scenarios. A careful observation of the probabilistic database tables tells us that the only possible answer to the above query is the tuple with seller id 239 since that is the only seller in the 12345 zip code. However, the output is 239 if and only if tuple  $z_1$  exists and at least one of the tuples  $y_1$  or  $y_2$  exist in the database. Therefore, we need to compute the probability of the boolean formula:  $(exists(y_1) \vee exists(y_2)) \wedge exists(z_1)$ , where  $exists(t)$  indicates the (random) variable corresponding to the event that tuple  $t$  exists. Complex queries may require us to compute the probability of non-trivial boolean expressions. For instance, if we want the list of reputed sellers offering Honda cars in the 12345 area, then, we will need to compute the probability of the boolean formula  $exists(x_1) \wedge (exists(y_1) \vee exists(y_2)) \wedge exists(z_1)$  – which is again an example of a *lineage* query. Note that more complex queries may generate several output tuples in which case we need to compute the probability of each of the output tuples. Computing the probability of the above formula requires us to reason about dependencies and correlations among all the variables in the (potentially large) formula efficiently, which is a challenging task.

The query evaluation model that we have illustrated so far requires us to compute the output probability distributions for each tuple and its associated



values. However, output results are not very intuitive to the users, who are interested to know why they actually obtained the result. Currently, if a user poses the query:  $Q$  : “List *reputed* car sellers in the *College Park* (12345) area who offer *Honda cars*”. Existing systems provide answers as shown in Figure 1.2(f) ( $Q$ ’s result shown). This output relation, however, is not very informative for the user since it does not provide any intuition about two important issues.

1. **(Explanations)** First, existing systems do not provide *explanations* for the query results – e.g., “Why is tuple  $t$  in the output result?” or “Why does output tuple  $t_1$  have such a high probability, as compared to tuple  $t_2$ ?”. As noted by Re et al. [92], even in a biological domain, where scientific decisions are made based on several uncertain hypotheses, it is critical to know the input hypotheses that contribute significantly to the output decision. Hence, it is essential to provide this information in addition to the actual results, to the users of the system.
2. **(Sensitivity Analysis)** Second, current systems do not provide the *sensitive* input tuples for a query, i.e., the set of tuples that can potentially modify the result probabilities the most. This information is critical in most application domains that need to handle uncertain data, because of the inexact nature of the probability values. The algorithms rely on probabilistic models such as Bayesian networks, CRFs and similarity logics to assign probabilities to the extracted tuples. Bayesian inference is #P-hard, therefore the probabilities computed are usually approximate (with no guarantees on bounds [25]), and further, similarity metrics are usually ad-hoc. Hence, the probabilities assigned to the tuples are often noisy, imprecise, and erroneous.

Hence, the users would be interested to know the input tuples that are likely to change the output probabilities significantly, when their probabilities are modified. In other words, we need to find the set of input tuples which highly *influence* the output probability values. Providing this information helps the user to focus his/her effort in *procuring more accurate probabilities* for that particular set of input tuples.

3. **(Re-evaluation)** Finally, since input tuple probabilities may be modified several times, the system also needs to solve the ensuing problem of supporting incremental updates to the query results by exploiting previously executed computation. For example, the users may choose to resolve uncertainty using data cleaning techniques based on Cheng et al. [14] or by using techniques based on value of information [68]. Or the users may procure more accurate values for the probabilities, e.g., by running more iterations [90] or by querying an expert.

### 1.1.3 Probabilistic Modeling of Data Streams [60]

Another real-world scenario where uncertain probabilistic data is generated is when *probabilistic modeling* is performed on data, especially in the context of continuously generated data streams. Enormous amounts of such streaming data are being generated everyday by measurement infrastructures that continuously monitor a variety of things from environmental properties using sensor networks [75] to behavior of large computational clusters [51]. To fully harvest the benefits of this extensive monitoring, there is a need to process and analyze such data streams in real-time. Probabilistic modeling of such streams is typically used for:

- 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 (e.g., *light* on a Berkeley Mote [34]). A common task over such data streams is to continuously *infer* the value of the hidden variables using the observed data.
- 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, poor coupling or analog-to-digital conversion, inaccuracies due to non-robust measurement techniques, 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.
- 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* [19, 83, 73]. For instance, Patterson et al. [83] 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.

All the tasks described above are examples of applying probabilistic models such as dynamic Bayesian networks [80, 79] to the raw data. These models allow us to combine prior domain knowledge about the system behavior with the actual observations to compute the most likely values for the variables being modeled. As a result of such probabilistic modeling, confidence/beliefs

are assigned to the raw measurements. For example, in the first task described above, the *hidden* variables cannot be inferred exactly, instead a probability distribution is associated with the hidden variable. Similarly, by removing measurement noise, we associate degrees of confidence with the measurements obtained by the sensor network. While extracting higher level events from low level sensor data, probabilities of occurrence of the higher level events are inferred. Hence, probabilistic modeling tasks generate highly correlated probabilistic streams. On these probabilistic streams, users and applications may be interested in several types of queries, as shown below:

1. **(Inference)** “What is the probability the temperature was more than 50 at 3pm today”. This is an inference query that requires us to compute the probability distribution of the temperature random variable.
2. **(Aggregation)** “What is the maximum temperature recorded by the sensor so far?”. This is an example of an aggregation query, that requires us to compute the probability distribution of the maximum temperature recorded by the sensor.
3. **(Sliding window aggregation)** “Specify the weekly average humidity measured by the sensor”. Sliding window queries are specific to streaming data sources. In this example, we need to compute the average humidity measured by the sensor in each week. Sliding window queries are much more challenging since we also need to capture the *temporal correlations* among the output values. In the above example, we need to capture the correlations between the humidity values in each week.

4. (**Lineage/Boolean formula**) “What is the probability that the temperature was below 50 today, increased to 75 the next day and was again below 50 the following day ?”. In this example, we need to compute the probability of observing the pattern.

As with the previous applications, the output of query evaluation is just the probability value, which provides little intuition to the user. Hence, we need to additionally provide more information pertaining to **explanations**, **sensitivity analysis** to the users.

## 1.2 Problem/Research Challenges

In this dissertation, we address the problem of query evaluation over large-scale uncertain data. In the following sections, we detail the main problems that we have worked on and the research challenges that we faced.

### 1.2.1 Probabilistic Modeling of Uncertain Data

As illustrated earlier, the first step involved in analyzing and querying uncertain data is to remove noise and infer hidden variables by using probabilistic modeling. Typical probabilistic modeling tasks use a wide range of models including Kalman filters, hidden Markov models (see Chapter 2 for a general overview) and other special purpose models. In order to support such modeling we not only need to develop declarative query languages using which the users can specify such complex models, but also develop algorithms that are general purpose, i.e., they can handle many kinds of probabilistic models. In addition, since the input data is in streaming fashion, our modeling algorithms

have to be very efficient, i.e., the time for modeling each item of the data is should be much smaller than the input data rate.

## 1.2.2 Query Processing over Correlated Probabilistic Data

The major focus of this dissertation is on efficient execution of a variety of declarative queries over large volumes of correlated uncertain datasets. In recent years, many probabilistic database systems have been developed to handle large-scale uncertain data [27, 95, 92, 5, 4, 96, 15]. While this prior work has made great strides in our understanding of how to manage large-scale uncertain data and how to evaluate the above types of queries on them, only a handful of these systems can handle correlated data. Sen et al. [96, 97] and Antova et al. [5, 4] have addressed issues in representing and querying complex correlations in probabilistic databases. However, their proposed techniques are not scalable to large databases. Letchner et al. [91, 71] developed techniques for processing simple queries over probabilistic event streams that have special correlation structure. However, many of their queries were limited to simple event detection queries (corresponding to 1 level boolean formulas). Also, their approach does not generalize to arbitrary correlations that we often find in real-world datasets. We illustrate the central challenge that is posed by presence of correlations in the following section. Here we first summarize the types of queries that we need to handle, and then discuss the challenges in executing them.

### Types of Queries:

1. Inference queries (also called What-If queries): As we explained in Sec-

tion 1.1.1, what-if queries require us to compute the probability distribution over a set of uncertain entities in the database. Inference queries may be an end unto themselves, but are more often are a precursor to further analysis.

2. Aggregation queries: Aggregation queries are very common in OLAP/Data Warehousing applications and they require us to compute the probability distribution of the aggregate value. A typical aggregation query is specified using SQL. Suppose we want to compute the number of Honda cars that are available for sale. We would issue the SQL query:

```
SELECT COUNT(*) FROM CarAds WHERE Model='Honda'
```

Other common aggregates that we need to compute are SUM, AVG, MIN and MAX.

3. Lineage queries: As illustrated in Section 1.1.2, conjunctive queries are quite common in applications that generate probabilistic data. Conjunctive queries are specified using SQL syntax. These are essentially SPJ (Select-Project-Join) queries and Select-From-Where queries which are restricted to equality joins and conjunctions in the WHERE clause. For instance, if we are interested to know the list of reputed auto dealerships near zip code 12345 offering Honda cars, we issue the SQL query:

```
SELECT CarAds.Seller
FROM CarAds, Location, Reputation
WHERE (CarAds.Seller = Location.Seller) AND (CarAds.Seller =
Reputation.Seller)
AND (Reputed = Good) AND (Location = '12345') AND (CarAds.Model
```

= 'Honda')

As explained in Sections 1.1.1 and 1.1.2, to evaluate the conjunctive query, we need to compute the probability of one or more boolean formulas (lineages), subject to the underlying set of correlations in the input database.

4. Extraction Queries: An Extraction query is a new type of query which we introduce here. The output of an extraction query is a small portion of the database that is relevant to the execution of the query and that affects the answer to the query. An extraction query is specified using a set of random variables and it is useful when we need to perform further query processing and analysis on the selected portion. Here, extracting a small portion of the database, while at the same time retaining all the correlation information, would be crucial for performance.
5. Sliding Window Queries: As illustrated in Section 1.1.3, sliding window aggregate queries are commonly posed in data stream management system. As mentioned in Section 1.1.3, we may be interested to monitor the sliding window average of temperature every week. We also consider a special case of a sliding window query, which we call as *tumbling window queries*, in which the length of the sliding window is equal to the skip length, e.g., weekly aggregates. In general sliding window queries are much harder than mere aggregates since we also need to capture the correlations in the output.



### 1.2.2.1 Challenges in dealing with Correlations

Correlations present in a probabilistic database make query evaluation challenging. We illustrate a few of these below.

1. There can be several types of correlations in a probabilistic database. A tuple can be correlated with another tuple (e.g., via mutual exclusion, cf. Section 1.1.2) or a set of tuples. Similarly, an attribute may be correlated with a set of other attribute values (e.g., temperature and humidity in a sensor network, cf. Section 1.1.3). This raises the first challenge: how to represent correlations in their most general form very efficiently.
2. The presence of correlations significantly influences the results of query evaluation. Computing the probabilities of output tuples when the base data tuples are correlated is a challenging task. Each tuple in a probabilistic database can be correlated with another tuple, or a set of tuples, which in turn may be correlated with several other tuples and so on. In a large probabilistic database, this correlation information corresponds to a very large graph (entities are connected based on correlations) where even searching for the relevant correlations in the graph can be quite inefficient.
3. Although the user specified correlations may be local in the graph (1-hop neighborhood), since the tuples are *mutually connected* to a number of other tuples in the graph, each tuple is correlated with every other tuple in the same connected component in the graph. This leads to the one of the most important challenges involved in supporting large-scale correlations, i.e., simple queries on few variables (2 or 3 variables) might

need to access the complete database. This is because two variables can be correlated with each other via a long chain of other random variables. To capture the joint probability between these variables we need to access the complete chain. However, this is a very expensive operation and we cannot afford to access the complete database each time a simple two variable query is posed. Hence, we need techniques for efficiently handling this issue.

4. The presence of correlations also influences the complexity of boolean formula queries. For instance, to compute the probability of a simple boolean formula  $(a_1 \vee a_2 \cdots \vee a_{100})$  exactly, where  $a_1, a_2, \dots, a_{100}$  are boolean random variables denoting the occurrence of events, we need to capture all possible dependencies that exist among these random variables efficiently. Specifically, we need to do this without constructing the full joint distribution among the set of random variables, which would be of size  $2^{100}$ , and subsequently infeasible to compute.
5. The presence of correlations significantly increases the query evaluation complexity for probabilistic data streams. Traditional query evaluation algorithms are typically not incremental and are hence not suitable for high throughput probabilistic streams. Hence, we need to develop efficient incremental algorithms for query evaluation.

### 1.2.3 Robust query processing: Sensitivity & Explanations

As we illustrated earlier, providing just output tuples along with their probabilities may not be intuitive to the user. Hence, we need to augment the query evaluation model with two additional features: namely *sensitivity analysis* and *explanations*. Providing such information makes it more intuitive to the user to reason about the query results.

**Sensitivity Analysis:** As described earlier, this requires us to determine the top- $\ell$  input tuples that most *influence* the results. Using sensitivity analysis, the users can focus their attention on procuring more accurate probabilities for these tuples.

**Explanations:** The explanations problem requires us to determine, for each query, the set of input tuples that most contribute to the answer. Note that the explanations problem is quite different from the sensitivity analysis problem. A tuple with high probability may contribute a high probability to the result (on account of its high probability), but may be very insensitive to the output probability value.

**Re-evaluation:** Since we allow the user to interactively modify input probabilities, we need to support fast re-evaluation of queries, i.e., faster than re-evaluating the query from scratch, preferably by exploiting previous computation.

## 1.3 Our Approach

To manage large-scale uncertain data, we first address the problem of modeling the uncertain data. As described in Section 1.1.3, there are three types of tasks that we typically need to do over uncertain data, i.e., inferring hidden variables, eliminating noise and discovering high level events from low level measurements. All the tasks described above are examples of applying probabilistic models such as dynamic Bayesian networks [80, 79] to the raw data. We use the abstraction of *model-based views* [35] to push the application of a wide range of probabilistic models to 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 probabilistic models and represent them as sets of weighted samples (called *particles*) in relational tables. Probabilistic models combine prior domain knowledge about the system behavior with the actual observations to compute the most likely values for the variables being modeled. As a result of such probabilistic modeling, confidence/beliefs are assigned to the raw measurements. For example, in the first task described above, the *hidden* variables cannot be inferred exactly, instead a probability distribution is associated with the hidden variable. Similarly, by removing measurement noise, we associate degrees of confidence with the measurements obtained by the sensor network. While extracting higher level events from low level sensor data, probabilities of occurrence of the higher level events are inferred. Hence, probabilistic modeling tasks generate probabilistic data. Further, owing to the temporal dependencies, the output probabilistic data is also highly correlated.

Next, we build a probabilistic database system for managing large-scale

correlated probabilistic data and develop algorithms for query processing over such data. To capture probabilistic databases in their most general form, we represent them using probabilistic graphical models (PGMs) [86, 24] developed in the machine learning community. PGMs allow us to represent joint probability distributions efficiently by exploiting the *independences* and the *conditional independences* in the data. The current literature on probabilistic database research [27, 95, 92, 5, 4, 96, 15] has considered the following two types of uncertainty in probabilistic databases.

1. *Tuple Uncertainty*: Here, there is uncertainty as to whether a tuple belongs to the database or not. For instance, the tuples in the Information Integration application (Figure 1.2(b)) had tuple uncertainty. We introduce a boolean random variable for each such tuple that takes value 1 when the tuple belongs to the table and 0 otherwise.
2. *Attribute Uncertainty*: Here, the value of an attribute is uncertain. For instance, in the probabilistic modeling application (Section 1.1.3), the value of the sensor temperature after the modeling process is complete is an uncertain attribute. To represent this, we introduce a random variable to denote the value of the attribute.

We capture correlations among these random variables by adding relevant edges between the correlated variables. The correlations are quantified by defining functions called *factors* over the random variables, that essentially *constrain* the values of the variables according to the correlation. We will discuss these issues in detail in Chapter 2. Using PGMs, we can capture all possible correlations that may be present in a probabilistic database. While the PGM representation is general, it is quite useful to use an alternative

representation of a PGM called a *junction tree* [40]. Queries posed on the probabilistic database can be seen as *inference* operations on the PGM as shown by Sen et al. [96]. We exploit this property for query processing.

Central to our technique is our proposed data structure, called *INDSEP*, which is an index data structure that provides orders of magnitude improvements in the query evaluation. INDSEP builds upon the well-known *junction tree* framework, designed to answer inference queries over large-scale probabilistic graphical models. Although such a junction tree over the probabilistic database can be adapted to answer queries directly (as we will discuss in Chapter 2), this naive approach can not avoid the critical problem mentioned above (Challenge 3, Section 1.2.2.1), and hence to answer a simple query, we may have to access and manipulate the entire junction tree. Our proposed INDSEP data structure provides the indexing support to answer these queries efficiently. In essence, the INDSEP data structure can be seen as a hierarchy of junction trees, each level subsuming the one below it, arranged in the form of an *n-ary tree* with appropriate *shortcut potentials* maintained at different levels of the index. The shortcut potentials are the key to the performance of INDSEP, using which we can answer queries in time logarithmic in the size of the database in most cases, depending on the correlation structure (as opposed to linear time or worse for the naive approach). Intuitively, the shortcut potentials allow us to skip over large portions of the database when computing joint distributions over variables that are correlated through long chains of other variables. Using INDSEP, we develop efficient algorithms for *inference* and *aggregation* queries.

For conjunctive query evaluation, we adopt a two-step process: In the first step, we determine the *lineage* of each output tuple, which is a boolean for-

mula that denotes the different possible derivations of the output tuple. In the second step, we evaluate the probability of the lineage over the junction tree corresponding to the probabilistic database. We show that, even for the restricted class of *read-once boolean formulas* (these are boolean formulas in which each term appears exactly once), the problem of evaluating the probabilities is  $\#P$ -complete. Hence, we propose a number of heuristics for this problem. In addition, we scale our algorithms to large probabilistic databases using the INDSEP data structure. For each output lineage, we initially precompute the complexity of evaluating its probability by using a novel quantity called *lwidth*, a quantity similar to the notion of graph treewidth [94]. We show that the complexity is exponential in the lwidth. For lineages with large lwidth, we develop approximation algorithms based on Monte Carlo techniques.

A peripheral module of our system is a unit for query processing over *probabilistic streams*, treated specially because of their ubiquity. We observe that although most real world probabilistic streams are highly correlated in both space and time, they obey a *highly structured* correlation structure. We observe that such streams are typically *Markovian*, with the same set of dependencies and independences repeating over time, i.e., the state at time “t+1” is independent of the states at previous times given the state at time “t” (in some cases, the state at time “t+1” may depend on a fixed number of states in the recent past [20]). Examples of such streams include the RFID data streams generated by the RFID Ecosystem application [103] and the RFID-based inventory management application [106]. Usually this is a result of the underlying physical process itself being Markovian in nature. In most applications, this is already encoded in the mechanism that generates the probabilistic streams; in our sensor network example (Section 1.1.3), the probabilistic stream would typi-

cally be generated by the application of dynamic Bayesian networks [60, 80] to sensor data, which by their nature generate Markovian and structured correlations. We exploit the knowledge of such structured correlations to efficiently evaluate queries over probabilistic streams. Firstly, we compactly encode the correlations in the Markovian stream by decoupling the correlation structure (the set of dependencies) from the probability values. Secondly, we develop techniques for incrementally processing several classes of queries. However, not all queries admit incremental evaluation; for such queries we provide polynomial time approximation strategies.

Finally, we develop a robust query processing framework for probabilistic databases by augmenting our probabilistic database system with support for performing low overhead *sensitivity analysis* and by providing *explanations* for query answers. Our system provides an option for the user to *mark* a query for performing either sensitivity analysis or explanation analysis over the results of the query. When a query is marked for sensitivity analysis, we provide the set of top- $\ell$  (where  $\ell$  is a user specified parameter) influential input tuples for the query and when a query is marked for explanations analysis, we provide the set of input tuples of size  $\ell$  which provides the best explanation for the query results.

## 1.4 Outline & Research Contributions

The outline and research contributions of the dissertation are as follows.

- In the next chapter, we begin by discussing prior related work. The research work described in the document relates and contributes to a number of different fields of research. We list the relevant references,



organized by the appropriate research area. In addition, we also provide background for the various concepts that we use throughout the dissertation. We describe our representation of probabilistic databases as probabilistic graphical models (PGMs). We also discuss the *junction tree* representation. Finally, we illustrate how to execute the queries posed over the probabilistic database directly over the junction tree representation.

- In Chapter 3, we provide an overview of the PrDB system that we have developed in the dissertation. We provide details regarding the *declarative* query language with which users can provide probabilistic data as input along with the correlations. We also discuss the data structures we use to store these correlations. We conclude with a discussion of the query processor.
- The first step in dealing with uncertain data is to apply probabilistic models over them. In Chapter 3.5 we develop algorithms for applying arbitrary probabilistic models over uncertain data streams. This chapter is based on our work which appeared in Kanagal et al. [60]. The contributions of this work include:
  1. We use the abstraction of *model-based views* [35] to push the application of a wide range of probabilistic models to streaming data inside a relational DBMS, thus enabling easy application of these tasks.
  2. We exploit the structure of *particle filters* (a widely applicable *sequential Monte Carlo* technique), to implement probabilistic mod-

eling and represent them using sets of weighted samples (called *particles*) in relational tables. This representation of DPMS naturally captures many of the correlations present in the data.

- After modeling the uncertain data using probabilistic models, we obtain correlated probabilistic database, which is represented using a *junction tree*. In Chapter 5, we introduce the *INDSEP* data structure that we build over the *junction tree* representation. This chapter is based on our work which appeared in Kanagal et al. [62]. Specifically, the research contributions of this chapter are:

1. We propose a novel hierarchical index structure, called *INDSEP*, for large correlated probabilistic databases, and introduce the idea of shortcut potentials which can result in orders of magnitude performance improvements.
2. We show how to answer inference queries, extraction queries and aggregation queries efficiently using *INDSEP*.
3. We develop algorithms for constructing a space-efficient index for a given database, using ideas developed in the *tree partitioning* literature. We also design techniques for keeping the index up-to-date in presence of updates.
4. We present a comprehensive experimental evaluation illustrating the performance benefits of our data structure.

- In Chapter 6, we provide algorithms to efficiently execute conjunctive queries over the probabilistic database using *INDSEP*. As stated earlier, we use a two step process in which we compute the *lineage* of the output

tuples as a boolean formula, in the first step and subsequently compute the probability of the boolean formula. The research contributions of this chapter are:

1. We develop a novel algorithm for computing the probabilities of boolean formulas over a *forest* of junction trees, a fundamental problem that has not been considered before.
  2. We show how to process a batch of lineages efficiently by exploiting the common subexpressions in the formulas.
  3. We devise Monte Carlo approximation algorithms for estimating the probabilities of boolean formulas over correlated junction trees.
- In Chapter 7, we consider *probabilistic data streams* that have a special *Markovian* correlation structure. We develop *incremental* algorithms for query processing over Markovian streams to enable scalable evaluation. The details in this chapter were originally published in Kanagal et al. [61]. The research contributions outlined in this chapter are:

1. We present an algebra for operating on probabilistic sequences (using the possible worlds semantics) and introduce the notion of Markovian sequences.
2. We develop efficient data structures for representing Markovian sequences and develop query processing techniques that exploit the repeated correlation structure.
3. We develop incremental algorithms for the query processing operators based on the `get_next()` framework to efficiently support streaming data.

4. We characterize queries that have exponential data complexity and provide approximation algorithms for them.
- In Chapter 8, we develop our robust query processing framework for probabilistic databases. We consider the case of tuple-independent probabilistic databases and a large class of queries including SPJ queries, aggregation queries and top-k queries. The main contributions of this chapter are:
    - We provide definitions for *influence* of tuple(s) on a query result and *explanations* that are applicable to a wide range of database queries including conjunctive queries, aggregation queries and top-k queries.
    - **Sensitivity Analysis**
      1. We show that the problem of identifying top- $\ell$  influential variables for conjunctive queries is #P-complete.
      2. For conjunctive queries that lead to *read-once / IOF* lineage formulas [98, 45, 81], we provide linear time algorithms (in size of the lineage). For the general case, we provide algorithms that are exponential in the *treewidth* [39] of the boolean formula.
      3. We develop algorithms for identifying influential variables for aggregation (SUM/ COUNT/ MIN/ MAX) queries.
      4. We develop *novel pruning rules* to speed up the computation of influential variables for top-k queries (by probability).
    - **Explanation Analysis**
      1. We show that the problem of determining the top- $\ell$  explanations for conjunctive queries is NP-hard in general. As with sensitivity

analysis, we develop algorithms for identifying the best explanations for queries that lead to read-once lineages.

2. We develop novel techniques for computing explanations for aggregation (SUM/ COUNT/ MIN/ MAX) queries.
- For conjunctive queries and aggregation queries, we provide *incremental* algorithms for re-evaluating query results when input probabilities are modified.

# Chapter 2

## Background and Related Work

In the first part of the chapter, we provide background for the various concepts used in the dissertation. Research in probabilistic modeling and probabilistic databases is highly multi-disciplinary with ideas from several areas in computer science ranging from machine learning, databases and graph theory being used. In the latter part of the chapter we discuss some of the related work in this area and provide references to the most relevant literature.

### 2.1 Background

As shown in Chapter 1, one way probabilistic databases are naturally generated is by applying probabilistic models to uncertain data. In this first part of the chapter, we provide background for *dynamic probabilistic models* (DPMs) such as *hidden Markov* models and *Kalman* filters. We illustrate inference algorithms over such models in Chapter 3.5.

Probabilistic databases can be equivalently represented using probabilistic graphical models (PGMs) [86], which is a useful tool developed to capture

uncertainty in the machine learning community. This result was originally shown by Sen et al. [96]. PGMs allow us to capture probabilistic databases in their complete generality. Any probabilistic database with tuple uncertainty, attribute uncertainty and arbitrary correlations can be represented as a probabilistic graphical model. In Section 2.1.2, we provide a brief overview of PGMs and illustrate how to represent a generic probabilistic database using PGMs.

Next, in Section 2.1.3, we provide algorithms for evaluating queries posed over the probabilistic database directly over its PGM representation. These algorithms are based on *inference* [86, 24] operations in PGMs. As we show later in the chapter, inference algorithms in PGMs do not scale to large probabilistic databases and it is more advantageous to use another equivalent representation of a PGM called a *junction tree* [40]. Junction trees, also known as *tree decompositions*, have been studied widely in several disciplines in computer science, including query optimization [10], constraint satisfaction [32], matrix decomposition algorithms and graph theory [94, 8, 7]. We describe the junction tree representation of PGMs in Section 2.1.4 and subsequently describe techniques for evaluating queries over junction trees in Section 2.1.5.

Finally, we discuss two special cases of probabilistic databases that have specialized correlation structures and allows for more efficient query evaluation. First, we introduce *Markovian streams*, i.e., probabilistic data streams which obey a *special* (linear) correlation structure, and discuss an alternative representation structure for Markovian streams. Second, we discuss *tuple-independent probabilistic databases*, which are essentially databases with only tuple uncertainty and no correlations.

### 2.1.1 Probabilistic Modeling of Uncertain Data

Dynamic probabilistic models are widely used in practice to model and to reason about complex real-world stochastic processes [56, 80, 79]. The simplest and most widely used examples of DPMS are *hidden Markov models (HMMs)* and *linear dynamical systems* (better known as Kalman filter models (KFM)). We start by illustrating HMMs and then describe more general DPMS. A more detailed illustration of DPMS can be found in the technical report [59].

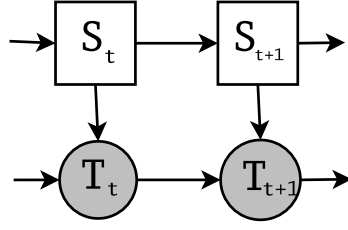
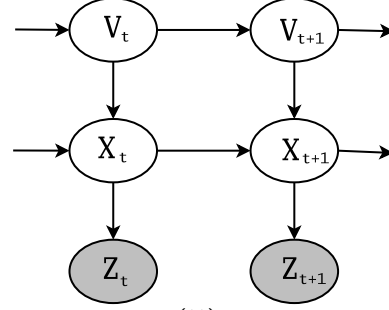
#### 2.1.1.1 Hidden Markov models (HMMs)

HMMs have been applied in a variety of areas like speech recognition, bioinformatics, and fault detection [88, 67, 112]. 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. We show the HMM in Figure 2.1(i). 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



(i) AR-HMM <sup>1</sup>

(ii) KFM

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

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

(iii)

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

(iv)

Figure 2.1: Graphical representations of DPMS. (i) Using an HMM for fault detection; (ii) Using a KFM for velocity and location estimation. (iii) Parameters of the HMM model. (iv) Parameters of the KFM model.

distributions (Figure 2.1(iii)) :

- $P(T_{t+1}|T_t, S_{t+1})$ : 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 2.1(iii) 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* or the *Viterbi algorithm* [88], 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).

### 2.1.1.2 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.

---

<sup>1</sup>AR-HMMs (Auto-Regressive HMMs) are a specific class of HMMs.

Figure 2.1(ii) shows a pictorial representation of the KFM that can be used in this application. (This model was described by Murphy [80]). 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 2.1(iv)).

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

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 [109]. 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 [72]. We will continue to call this model the *Kalman filter model* (KFM).

### 2.1.1.3 DPMs: Graphical Representation

DPMs generalize the basic models described above. They are represented using a directed graphical structure (Figure 2.1), where the graph captures the dependencies between the process variables. Figure 2.1(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 6.4). 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 2.1(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 2.1 (iii).

**Edges/CPDs:** The directed edges represent “causality”. In Figure 2.1(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 2.1(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 [59].

#### 2.1.1.4 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 4.3.

### 2.1.2 PGM Representation

In this section, we describe how to represent a probabilistic database generically using a PGM. Consider a simple probabilistic database on two relations  $R_1$  and  $R_2$  shown in Figure 2.2(a). The relation  $R_2$  exhibits attribute uncertainty, i.e., both the attributes  $V_1$  and  $V_2$  are uncertain. The relation  $R_1$  exhibits both attribute uncertainty in the attribute  $V_0$  and tuple uncertainty since it has a column marked ‘?’. The random variables corresponding to the uncertain entities are also shown in the figure. For instance, the tuple with  $gid = 2, (2, h, f)$  in  $R_2$  has two random variables  $h$  and  $f$  that denote the values of the uncertain attributes  $V_1$  and  $V_2$ . Similarly, the tuple  $(1, b)$  in  $R_1$  denotes the value of the attribute  $V_0$  using the random variable  $b$ . In addition,

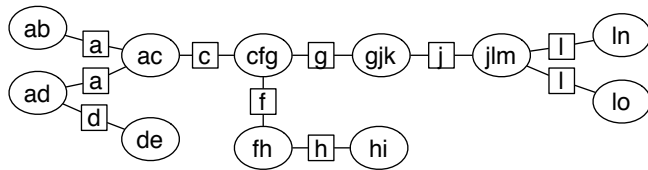
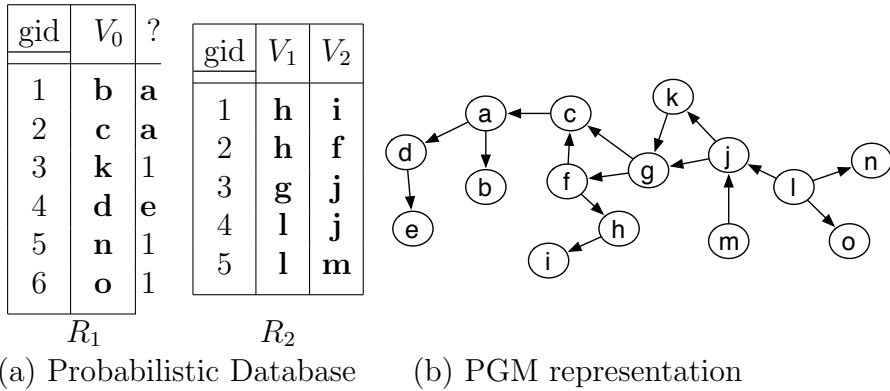


Figure 2.2: Example: (a) A probabilistic database  $D^p$  on two relations  $R_1$  and  $R_2$  exhibiting both tuple-existence and attribute-value uncertainties (e.g.  $a$  indicates the random variable corresponding to the existence of the first tuple in  $R_1$ ); (b) The directed PGM that captures the correlations in  $D^p$ , and (c) the junction tree representation of the PGM.

to denote the tuple uncertainty, we use a binary random variable  $a$  to denote the presence of the tuple, i.e., if  $a = 1$ , then the tuple belongs to the relation, otherwise, if  $a = 0$ , then the tuple does not belong to the relation. Notice the perfect correlation between tuples 1 and 2 in  $R_1$ , i.e., either both of them belong to the database together, or neither of them appears, depending on the value of the random variable  $a$ . Further, many more complex correlations can occur among the random variables. We represent these correlations using a graphical structure as shown in Figure 2.2(b). The PGM in the figure is an example of a *Bayesian Network*, a class of directed probabilistic graphical models. Although our system is not restricted to directed PGMs, we use the example of directed graphical models for simplicity of exposition. Figure

2.2(b) depicts a directed PGM on a set of random variables  $\{a, b, \dots, o\}$  corresponding to the probabilistic database shown in Figure 2.2(a). Every node  $v$  in the PGM is associated with a conditional probability distribution  $P(v|Pa(v))$  ( $Pa(v)$  is the set of parents of  $v$ ), which denotes how the value of  $v$  depends on the values of its parents. For example, node  $g$  in Figure 2.2(b) is associated with the conditional probability distribution  $p(g|k, j)$  since the parents of  $g$  are  $k$  and  $j$ ; similarly, the node  $f$  is associated with the conditional probability distribution  $p(f|g)$ . Nodes with no parents have prior marginal probabilities attached to them. In Figure 2.2(b), the nodes  $l$  and  $m$  have no parents and are associated with the prior probability functions  $p(l)$  and  $p(m)$  respectively. The overall joint distribution over all the variables can be computed by multiplying the conditional probability distributions of all the nodes in the PGM as shown in the equation below.

$$p(a, b, \dots, o) = (p(m)p(l)p(n|l)p(o|l)p(j|m, l)p(g|k, j)p(k|j)p(c|f, g) \\ p(f|g)p(h|f)p(i|h)p(a|c)p(b|a)p(d|a)p(e|a))$$

Missing edges in the graph encode the conditional independences between the random variables. For example, in Figure 2.2(b),  $e$  is independent of  $a$  if we know the value of the random variable  $d$ . Similarly,  $b$  is independent of  $c$  if we know the value of the random variable  $a$ .

### 2.1.3 Query Processing over PGMs

In this section, we describe how to execute queries on probabilistic databases represented as PGMs. We use the example of a simple PGM shown in Fig-

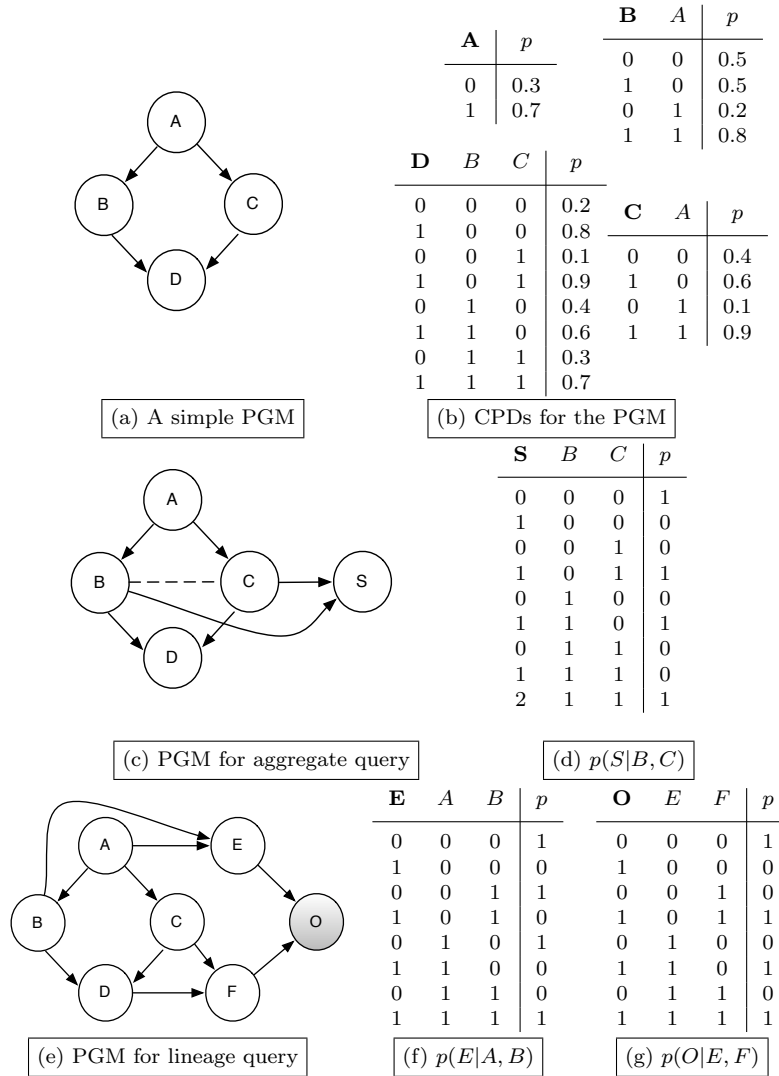


Figure 2.3: Query processing over probabilistic databases using graphical models: (a) a graphical model over 4 attributes; (b) an example set of CPDs for the graphical model (bold-faced variables indicate the child nodes); (c, d) to execute a query over the probabilistic database, we add new variables to the PGM and introduce additional CPDs; (e) PGM for evaluating the lineage query (f) AND factor for variable  $E$ ; (g) OR factor for variable  $O$

Figure 2.3 to illustrate the query processing algorithms. Figure 2.3 depicts a PGM on four binary valued random variables  $A$ ,  $B$ ,  $C$  and  $D$ . According to this model, the random variable  $D$  is conditionally independent of  $A$  given the *value* of the random variables  $B$  and  $C$ . The CPDs for our example graphical



model are shown in Figure 2.3(b).

To evaluate a query over such a database, Sen et al. [96] propose adding new random variables to the graphical model to capture the intermediate result tuples generated during query processing. We consider the following queries.

1. Inference query: As indicated earlier (Chapter 1), this query requires us to compute a joint probability distribution or a conditional probability distribution among a set of random variables. For instance, suppose we want to compute the probability distribution of random variable  $D$ , i.e.,  $p(D)$ . For this query we consider the original PGM (Figure 2.3(a)) and eliminate the non query variables  $B$ ,  $C$  and  $D$  using the same inference algorithm described above. To compute conditional distributions such as  $P(D|A)$ , we first compute the joint distribution  $p(A, D)$  and subsequently convert this into a conditional distribution by dividing by  $p(A)$ .
2. Aggregation query: We illustrate aggregation with an example. Suppose that we would like to determine the sum of random variables  $B$  and  $C$  in the PGM. To compute the sum, we introduce a new random variable  $S$  in the PGM that represents the sum of  $B$  and  $C$ , and we add edges from  $B$  and  $C$  to  $S$  since they influence the value of  $S$ . The modified PGM is shown in Figure 2.3(c). The exact dependence itself is captured using the CPD  $p(S|B, C)$  shown in Figure 2.3(d); the CPD encodes the fact that  $S$  is the sum of  $B$  and  $C$ . Now, the query evaluation problem is reduced to the computation of the *marginal distribution* of the random variable  $S$ .

Computing the marginal distribution is a well studied problem on PGMs called *inference*. There are a number of algorithms for performing in-

ference on PGMs such as variable elimination [113], belief propagation [86] etc. We illustrate variable elimination using the above example. To determine the marginal distribution of node  $S$ , in essence we need to perform the following computation:

$$p(S) = \sum_{A,B,C,D} p(A, B, C, D, S)$$

From the joint distribution of the random variables, we need to *sum out* (eliminate) the variables that we do not require, in this case  $A$ ,  $B$ ,  $C$  and  $D$ . The variable elimination algorithm takes in the order of elimination as input and sums out the variables in the order specified. The first two steps of the elimination (to eliminate  $A$ ) are as follows:

$$\begin{aligned} p(S) &= \sum_{B,C,D} p(D|B, C)p(S|B, C) \underbrace{\sum_A p(A)p(B|A)p(C|A)}_{f(B, C)} \\ &= \sum_{B,C,D} p(D|B, C)p(S|B, C) f(B, C) \end{aligned}$$

The order of elimination affects the complexity of the computation. A bad ordering can potentially result in an exponential computation (in the number of nodes in the graph) while a good ordering can make inference polynomial-time computable. Some PGMs may not have a good ordering at all, in which case, the inference problem is *#P-hard*. We can visualize the reasons for such scenarios by observing the changes that occur to the PGM while eliminating variables. In the above example, when  $A$  is summed out from the expression, a new dependency between  $B$  and  $C$  is created, which is quantified by the function  $f$  in the equation. In other words, eliminating  $A$  introduces an *edge* between  $B$  and  $C$  in the graph.

In general, edges are introduced between *every pair of neighbors* of the node that is being eliminated. If during the elimination process, a node gets connected to a large fraction of the other nodes in the graph, it would result in the creation of a very large joint probability distribution (possibly exponential in the number of nodes in the graph) since the dependencies between all the nodes need to be captured.

3. Lineage query: We illustrate this with the same PGM as shown in Figure 2.3. Suppose that we want to compute the probability of the boolean formula  $(A \wedge B) \vee (C \wedge D)$ . As with the aggregation query, we introduce additional random variables to denote the intermediate results of the query. Here, we introduce three new random variables  $E$  and  $F$  and  $O$  as shown in Figure 2.3(e). Here,  $E = A \wedge B$  and  $F = C \wedge D$ . Finally, the output  $O$  is given by  $O = E \vee F$ . The exact dependence is captured using *truth tables* for the AND and OR logic. For example, the factor for  $p(E|A, B)$  is shown in Figure 2.3(f) and the factor for  $p(O|E, F)$  is shown in Figure 2.3(g). As before, query evaluation is reduced to the problem of computing the marginal distribution for the variable  $O$ .

#### 2.1.4 Junction Tree Representation of PGMs

A PGM can be equivalently described using a *junction tree* representation [40], also commonly known as a *clique tree* in the graph theory literature. We refer the reader to Darwiche et al. [52] for full details involved in the construction of a junction tree and only discuss this briefly here. Constructing the optimal junction tree for a PGM has been shown to be NP-hard [40]. Building a junction tree consists of 3 main steps explained below. We illustrate

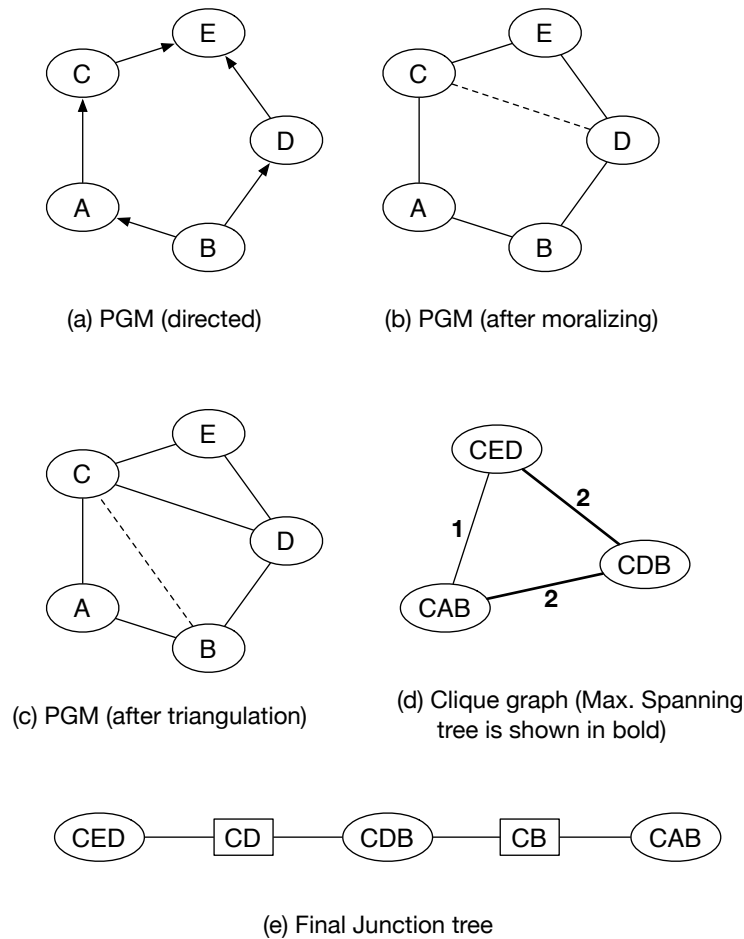


Figure 2.4: Figure shows the various steps in the construction of the junction tree from the original PGM representation in part(a). In part(b) we moralize the PGM by “marrying” (connecting) nodes  $C$  and  $D$ . In part(c) we triangulate the resulting graph by adding an edge between nodes  $C$  and  $B$ . Note that an edge between  $A$  and  $D$  would also work. In part(d), we construct the clique graph. The weight of each edge is indicated and the maximum spanning tree is also shown. The final junction tree is shown in part(e).

with an example in Figure 2.4.

1. Moralizing the PGM: This step is required for directed PGMs. It involves 2 steps. Suppose that we start with a directed PGM denoted by  $G$ . In the first step, we create a copy of  $G$  denoted by  $G_u$  which is an undirected version of  $G$  simply by dropping the direction from the edges.

Next we create the moralized graph  $G_M$  as follows. For each vertex  $v$ , we identify its parents  $\Pi(v)$  in  $G$ . Then, we connect all the nodes in the set  $\Pi(v)$  with each other in  $G_u$ . The resulting graph is the moralized graph  $G_M$ . In Figure 2.4, the PGM in 2.4(a) is moralized by adding an edge between nodes  $C$  and  $D$ .

2. Triangulating the moralized PGM: An undirected graph is triangulated if and only if every cycle of length greater than or equal to 4 has a chord (edge that connects non-adjacent vertices). The minimum triangulation problem is known to be NP-complete [111]. Hence, we will use a heuristic proposed by Kjaerluff et al. [64] which we describe briefly. We go through the nodes of  $G_M$  and for each node  $v$ , we select its neighbors and add additional edges to convert them into a clique. Clearly, the ordering of the vertices influences the number of additional edges added. The heuristic here is to pick the next node such that we add *as few* additional edges as possible, breaking ties by choosing the node that introduces the smallest clique. In Figure 2.4, we triangulate the graph by adding an edge between nodes  $C$  and  $B$  as shown in 2.4(c).
3. Building the junction tree: Next, we identify the maximal cliques in the graph. Maximal cliques are those cliques which are not properly contained in a larger clique. Next we create a new *clique graph* by introducing a new vertex for each maximal clique. We introduce edges between the cliques that share a common vertex in the original graph  $G_M$ . The edges are assigned weights equal to the number of vertices they have in common. The junction tree is obtained by constructing a maximum spanning tree of this graph. A maximum spanning tree ensures the *run-*

*ning intersection property* of the junction tree, which is discussed below.

The clique graph and the junction tree for our running example is shown in Figure 2.4(d,e).

The nodes in the junction tree which correspond to the maximal cliques in the undirected PGM are called *clique nodes*. In addition, for each edge in the tree, we create a new node, called a *separator node* which corresponds to the *set intersection* between the adjacent clique nodes. We connect this new node to the end points of the original edge. Separator nodes correspond to the cut vertex sets that separate the maximal cliques in the undirected PGM. A junction tree satisfies the *running intersection property*: for a variable  $v$ , if  $v \in C_1$  and  $v \in C_2$ , then  $v$  is present on all the cliques and separators in the path joining  $C_1$  and  $C_2$ . After the tree is constructed, we assign each of the conditional and prior probability distributions corresponding to the nodes of the PGM into a relevant clique in the junction tree. We then multiply all the probability distributions within a single clique and store it in the clique as its *clique potential*. Following this, we run a *message passing* algorithm [52] on the tree, during which the cliques locally transmit information about their distributions (also called *beliefs*) to neighboring cliques, which the neighboring cliques use to modify their potentials; the neighboring cliques in turn send their beliefs to their neighbors and so on. Typically the process is started with choosing a *pivot* node from which the messages are first sent outward, and then collected back inward. After this step, the clique potential of a clique node will be equal to the *joint distribution* of all the variables present in the clique. Similar condition applies to the separator nodes as well.

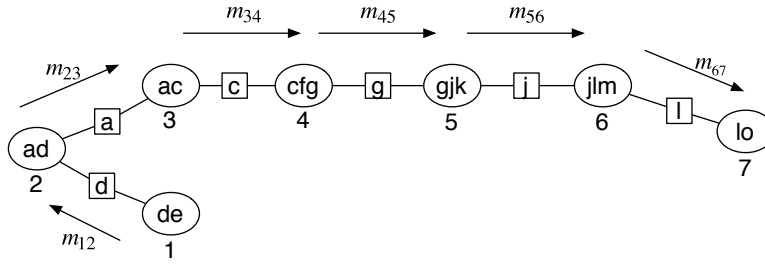


Figure 2.5: Path constructed for query  $\{e,o\}$

The overall joint distribution represented by the junction tree can be computed as follow. Suppose we denote the joint probability distribution of clique  $C_i$  by  $p(C_i)$  and that of separator  $S_k$  by  $p(S_k)$ . Then the overall joint distribution is given by the equation:

$$p = \frac{p(C_1)p(C_2)\dots p(C_n)}{p(S_1)p(S_2)\dots p(S_k)}$$

which is basically the product of all the clique potentials, divided by the product of all the separator potentials. For the junction tree shown in Figure 2.2(c), the value of the joint distribution is:

$$\frac{p(ab)p(ac)p(ad)p(de)p(cfg)\dots p(ln)p(lo)}{p(d)p(a)^2p(c)\dots p(l)^2}$$

### 2.1.5 Query Processing over Junction Trees

We now provide algorithms for query processing over junction trees. As indicated earlier, we consider the four types of queries specified in Section 1.2.2.

**Extraction Queries:** The output of an extraction query in the context of the junction tree representation is defined (informally) as a junction tree that includes all the query variables and all the correlations that exist among the

variables. In other words, we are interested in extracting the most relevant part for the query variables from the huge junction tree. Extraction queries are useful when we need to perform further query processing and analysis on the selected random variables. Here, extracting a small portion of the junction tree, while at the same time retaining all the correlation information, would be crucial for performance.

A naive algorithm to execute an extraction query is by computing the smallest *Steiner tree* on the junction tree that connects all the query variables of interest. Note that a Steiner tree can be computed on a tree structured graph in polynomial time. Consider an extraction query  $\{g, k\}$  on the junction tree in Figure 2.2(c). On examining the junction tree, we find that the clique  $g, j, k$  contains both the query variables  $g$  and  $k$ . Hence the output to this query is just the clique  $gjk$ . Note that since the clique contains the joint distribution of  $gjk$ , it encodes all the correlations between  $g$  and  $k$ . Now consider an extraction query  $\{e, o\}$  on the same junction tree. For this case, we observe that  $e$  is contained in clique  $de$  and  $o$  is contained in clique  $lo$ . The Steiner tree for this query reduces to a simple path, which is shown in Figure 2.5. If a query variable is present in multiple cliques, then we choose the clique that reduces the overall size of the Steiner tree. This can be performed as a post processing operation after computing the Steiner tree by exploiting the running intersection property – we can remove a leaf node from the Steiner tree if its neighbor (in the tree) has all the query variables present in the leaf node.

We note that the answer to an extraction query is not unique – in fact, a major focus of our work here is developing a technique that efficiently extracts the smallest possible junction tree that still captures all the correlations among



the query variables.

**Inference Queries:** The output of an inference query is the joint distribution over all the variables present in the set. The answer to a what-if query can be computed by executing the inference query and later conditioning it to obtain the required conditional distribution.

To execute an inference query, we first run the extraction query over the set of variables and obtain a junction tree over them. We then execute the HUGIN algorithm [24] for computing the required joint distribution. We illustrate this with two examples. Consider the inference query  $\{g,k\}$ . After executing the extraction query, we receive the clique  $gjk$  as shown above, and then simply *eliminate (sum out)* the variable  $j$  from the joint distribution  $p(g, j, k)$  and return  $p(g, k)$  to the user. In other words, we compute:

$$p(g, k) = \sum_j p(g, j, k)$$

Now consider the inference query  $\{e,o\}$ . As before, we run the extraction query and obtain the path shown in Figure 2.5. Using such a path, we can compute the joint distribution over all the variables present in the path using the formula discussed above, following which we can eliminate the non-query variables and determine the answer,  $p(e, o)$ . However, the intermediate joint distribution computed will be extremely large. We can instead execute the query more efficiently by eliminating the unnecessary variables early on using message passing. We now show the sequence of steps for determining  $p(e, o)$ . We first establish the direction of message passing and the pivot node – the node to which all the messages are sent. In this example, we assume that the pivot is node  $lo$ , and the messages are sent along the path from  $de$  to  $lo$  as

shown in Figure 2.5. In the first step, clique  $de$  sends a message  $m_{12}$  (See Figure 2.5) to clique  $ad$  which is basically the value of the joint distribution  $p(d, e)$ . After receiving this message, the clique  $ad$  multiplies the message with its potential  $p(a, d)$  and divides by  $p(d)$  to obtain the joint distribution  $p(a, d, e)$ . However, since  $d$  is not required for future computation, it eliminates  $d$  from this distribution to determine the probability distribution  $p(a, e)$ . The clique  $ad$  sends message  $m_{23} = p(a, e)$  to clique  $ac$  to continue the message passing. Note that  $e$  is needed since it is part of the query variables and also that  $a$  is required for correctness of the algorithm since it appears in the next edge. Each clique determines the variables that are necessary by looking at the neighbor to which it has to send a message and the set of query variables. Once clique  $ac$  receives message  $m_{23}$ , it uses its potential  $p(a, c)$  to determine the joint distribution  $p(a, c, e)$  and then eliminates  $a$ , generating message  $m_{34} = p(c, e)$ . This process is continued until we reach the clique  $lo$  at which point, we eliminate all the non-query variables and determine the value of  $p(e, o)$ .

**Aggregation Queries:** For computing aggregation queries, we perform the extraction query and obtain a small junction tree on the relevant variables from the underlying junction tree. We can then construct the appropriate graphical model for the aggregation function and use an inference algorithm as described by Sen et al. [96] for computing the probability distribution of the aggregate value. We discuss aggregate query evaluation in more detail in Section 5.3.2.

**Lineage Queries:** Lineage queries are important in the context of probabilistic databases. Lineage or Provenance [95] of a tuple in a database is a boolean

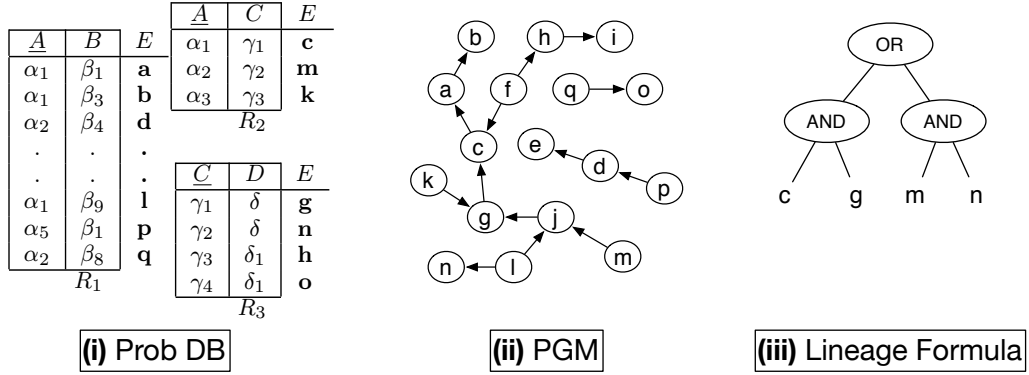


Figure 2.6: Figure shows (i) a tuple uncertain probabilistic database (ii) the graphical model that captures the correlations among the various tuples

formula, which represents all possible derivations of the tuple. Suppose that we want to execute query  $R = \Pi_D(R_2 \bowtie_C R_3)$  over the database in Figure 2.6(i). Note that all the relations in the database have tuple uncertainty, but no attribute uncertainty. For simplicity of exposition, we only consider lineage queries in the context of tuple uncertainty probabilistic databases. The correlation among the random variables in the database is shown in Figure 2.6(ii). Consider the tuple  $(\delta) \in R$ . It is generated by the projection of tuples  $(\alpha_1, \gamma_1, \delta)$  and  $(\alpha_2, \gamma_2, \delta)$  which are present in  $(R_2 \bowtie_C R_3)$ . If either of the tuples are present in the join, then the output will contain  $(\delta)$ . Hence, the lineage of  $(\delta)$  is written as the boolean OR of the lineages of the tuples  $(\alpha_1, \gamma_1, \delta)$  and  $(\alpha_2, \gamma_2, \delta)$ .  $\lambda(\delta) = \lambda(\alpha_1, \gamma_1, \delta) \vee \lambda(\alpha_2, \gamma_2, \delta)$ . The tuple  $(\alpha_1, \gamma_1, \delta)$  itself is dependent on the presence of both tuples  $(\alpha_1, \gamma_1)$  and  $(\gamma_1, \delta)$  in relations  $R_2$  and  $R_3$  respectively. Hence, the lineage of  $(\alpha_1, \gamma_1, \delta)$  is written as the boolean AND of the tuples  $(\alpha_1, \gamma_1)$  and  $(\gamma_1, \delta)$  as:  $\lambda(\alpha_1, \gamma_1, \delta) = \lambda(\alpha_1, \gamma_1) \wedge \lambda(\gamma_1, \delta) = c \wedge g$ . Similarly,  $\lambda(\alpha_2, \gamma_2, \delta) = m \wedge n$ . Hence, we can write the overall lineage of the output tuple  $(\delta)$  as the following boolean formula  $((c \wedge g) \vee (m \wedge n))$ . This formula is an example of a *read-once* boolean formula [45], i.e., each boolean

variable appears exactly once in the formula. It can be represented as a parse tree, as illustrated in Figure 2.6(iii). The root node of the tree corresponds to the entire boolean formula. Intermediate nodes correspond to subformulas, i.e., they represent the formula of the subtree below them, e.g., in Figure 2.6(iii), the intermediate node 1 corresponds to  $(c \wedge g)$  and the node 2 corresponds to  $(m \wedge n)$ .

A lineage query requires us to evaluate the probability of the lineage formula given the probability distribution of the input variables. A naive method is to use the joint probability distribution over the variables in the lineage. Suppose we need to compute the probability distribution of  $(a \wedge b)$ . We first compute the joint pdf over the variables  $a$  and  $b$ , i.e.,  $p(a, b)$ . Then, we use the *conditional distribution*,  $p(a \wedge b | a, b)$  which specifies how the random variable  $a \wedge b$  depends on  $a$  and  $b$ . In this case, it is just the truth table of the boolean AND logic. We multiply  $p(a, b)$  with  $p(a \wedge b | a, b)$  and obtain  $p(a, b, a \wedge b)$ . Following this, we eliminate  $a$  and  $b$  from  $p(a, b, a \wedge b)$  and determine  $p(a \wedge b)$ . Note that  $a \wedge b$  is just another boolean variable with domain  $\{0,1\}$ .

**Expressions:** We collectively refer to lineages (e.g.,  $b \wedge c, d \wedge e$ ) and singleton random variables (e.g.,  $a, f$ ) as *expressions*. As we describe later, we will often need to compute the joint probability distribution of a set of expressions, e.g.,  $\{a, b \wedge c, d \vee e\}$  – we call them *ExpressionSets*.

The algorithm described above does not scale to large expressions since the initial step computes a huge joint distribution. Even a simple formula of size 25 needs to compute a joint pdf of size  $2^{25}$ , which is very inefficient. We develop better algorithms for processing lineages over junction trees, based on message passing in Chapter 6.

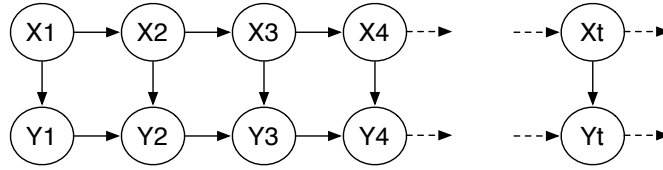


Figure 2.7: Example of a Markov sequence

### 2.1.6 Special Case 1: Markovian streams

So far, we have taken a look at representing a generic probabilistic database with arbitrary correlations. We can extend the PGM and the junction tree representation to probabilistic data streams as well. However, as we observed, most real world data streams obey very structured correlations with the same set of dependences and independences repeated across time. These data streams obey the Markovian conditional independence property, i.e., the random variables corresponding to the tuple at time  $t - 1$  is independent of those at time  $t + 1$ , given the value of the tuple at time  $t$ . An example of a Markovian stream is shown in Figure 2.7. As we can see from the figure, the dependencies repeat over time, i.e., for all values of  $i$ ,  $X_i$  influences  $Y_i$  and  $X_i$  influences  $X_{i+1}$ . Similarly for all values of  $i$ ,  $X_{i+1}$  and  $Y_i$  influence  $Y_{i+1}$ . Also note the *Markovian* property: Given the values of  $X_i$  and  $Y_i$ ,  $X_{i+1}$  and  $Y_{i+1}$  are independent of all the previous random variables. We will discuss how to represent and query these probabilistic streams efficiently in Chapter 7.

### 2.1.7 Special Case 2: Tuple Independent Probabilistic Databases

In this section, we provide background for tuple uncertainty probabilistic databases and the various concepts discussed in Chapter 8. In a tuple un-

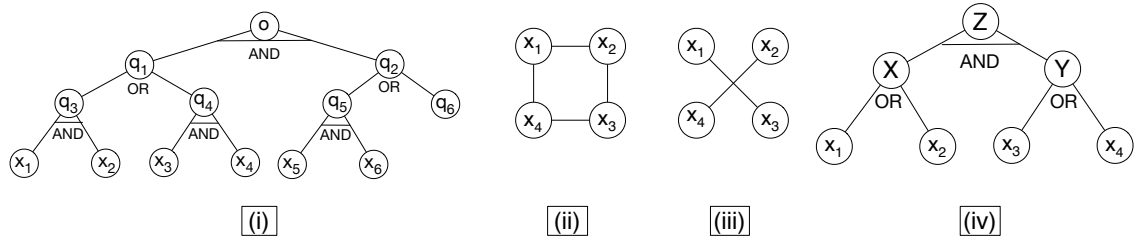


Figure 2.8: (i) Boolean formula  $(x_1x_2 + x_3x_4)(x_5x_6 + x_7)$  represented using an AND/OR tree. Leaves denote variables of the formula, internal nodes are intermediate expressions. (ii, iii, iv) Steps involved in generating the read-once formula for  $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4) \vee (x_4 \wedge x_1)$

certainty probabilistic database, (Dalvi et al. [27]), each tuple  $t_i$  exists with probability  $p_i$ . Also, the existence of a tuple is independent of the existence of the other tuples in the database. We associate each tuple  $t_i$  in the database with a *binary random variable*  $x_i$  such that  $x_i = 1$  if  $t_i$  belongs to the database and  $x_i = 0$  otherwise. Note that  $p_i = \Pr(x_i = 1)$ . Let  $P = \{p_1, p_2, \dots, p_n\}$  be the set of input probabilities.

### 2.1.7.1 Queries

In this section, we consider conjunctive queries, top-k queries (by probability) and aggregation queries. We describe each in turn.

**Conjunctive queries:** A conjunctive query is a fragment of first-order logic restricted to  $\exists$  and  $\wedge$ . In SQL, they correspond to **select-project-join** queries, restricted to equi-joins and conjunctions in the **where** clause. As shown by Das Sarma et al. [95], a conjunctive query can be evaluated over a probabilistic database by first computing the *lineage/provenance* for each output tuple, which is a boolean formula that represents all possible derivations of the output tuple, and subsequently evaluating the probabilities of the lineage

formulas. The general problem of conjunctive query evaluation has been shown to be #P-complete [27]. However, if the lineage formula can be represented using a *read-once* or 1-OF form [81, 98] (a boolean formula in which each literal appears exactly once), then the probability of the lineage can be computed in polynomial time. For example, the boolean formula  $(x_1 \wedge x_2) \vee (x_2 \wedge x_3)$  can be rewritten as  $x_2 \wedge (x_1 \vee x_3)$  which is a read-once formula. On the other hand,  $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_1)$  cannot be rewritten as a read-once formula. A read-once formula can be represented as an *AND/OR* tree as shown in Figure 2.8. The probability of a read-once formula can be computed using a bottom up algorithm over the AND/OR tree, which computes probabilities of intermediate nodes using the following equations. Suppose that  $z$  is a node with children  $x_1$  and  $x_2$ :

$$\text{if } z = x_1 \wedge x_2 \qquad p(z) = p(x_1)p(x_2) \qquad (2.1)$$

$$\text{if } z = x_1 \vee x_2 \qquad p(z) = 1 - (1 - p(x_1))(1 - p(x_2)) \qquad (2.2)$$

Given a boolean formula in DNF form, we can check whether it can be represented using a read-once representation in time linear in the size of the formula using the *co-graph recognition* algorithm of Golumbic et al. [45]. We illustrate this algorithm with an example in Section 2.1.7.2. If the boolean formula cannot be represented in a read-once format, we use *Shannon expansions* as described by Olteanu et al. [81] in order to compute its probability (See Section 2.1.7.3).

**Probabilistic Threshold/Top-k Queries:** We consider:

1. Probabilistic threshold queries: A conjunctive query  $Q$  is specified along with a threshold probability value  $\tau$ ; the output is the *set* of the output tuples

of  $Q$  whose probabilities exceed  $\tau$ . We assume that the output probabilities are not part of the result.

2. Top-k queries by probability: This query requires us to return the set of top-k tuples sorted by probability. This is different from threshold queries because in this case, we only return  $k$  tuples, whereas threshold queries are not restricted to  $k$  output tuples.

**Aggregation Queries:** The final class of queries we consider are aggregation queries such as SUM, MIN, MAX and AVG. In this chapter, we only consider aggregates over a single table containing a set of independent base tuples. Each base tuple has a real valued score attribute  $A$  and the above aggregation functions operate on the scores. We use  $a_i$  as a shorthand notation for  $t_i.A$ , the score of tuple  $t_i$ . For both the sensitivity analysis problem and explanation analysis problem, we consider the *expected values* version where the output of the query is the expected value of the aggregate. For example, the answer to SUM is  $\mathbb{E}[\sum_i a_i x_i]$ , where  $x_i$  is the binary random variable defined in Section 2.1.7.

### 2.1.7.2 Detecting read-once lineages

In this section, we describe Golumbic’s algorithm [45] which takes as input, a DNF boolean expression  $\lambda$  and determines if  $\lambda$  can be rewritten using a read-once representation. It also returns the read-once rewriting if it exists. The algorithm is *complete*, i.e., if it cannot determine a rewriting, then it does not exist. The algorithm starts by constructing a *co-occurrence graph* of the boolean formula. The co-occurrence graph of the formula is an undirected graph in which the nodes are literals of the formula and an edge exists between



2 literals if they occur together in some clause in the DNF formula. In the next step, the algorithm checks if the co-occurrence graph is a *co-graph* [23]. If yes, then the algorithm computes the *co-tree* representation of the co-occurrence graph, which is the required read-once representation. We illustrate the algorithm with an example.

Suppose we have a boolean formula:  $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4) \vee (x_4 \wedge x_1)$ . The co-occurrence graph  $G$  for the formula is shown in Figure 2.8(ii). Since the graph is connected, the algorithm creates an AND node ( $Z$ ) in the co-tree (Figure 2.8(iv)) and constructs the complement of  $G$ ,  $G^c$  as shown in part(iii). If  $G^c$  is connected, then the formula does not have a read-once representation. Otherwise, it considers each component separately. It creates an OR node in the co-tree for each component, which are made the children of ( $Z$ ). The algorithm recursively continues until we reach singleton graphs, which are added as leaves, or we reach a termination due to non-existence of a read-once representation.

A small caveat with this approach is that the algorithm works only for *normal* boolean formulas. A boolean formula is normal if we can reconstruct it from its co-occurrence graph. For example,  $x_1 \wedge x_2 \wedge x_3$  is normal but  $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4)$  is not normal. Note that the co-occurrence graphs for both formulas are the same. In recent work, Sen et al. [98] have shown that lineage formulas are always normal (for conjunctive queries without self-joins), hence we do not need to make this check. Next, we discuss Shannon expansions of a boolean formula, which we use for handling non-read-once lineages.

### 2.1.7.3 Shannon Expansions

Shannon expansions is a technique for representing a boolean formula as a XOR of two smaller sub-functions of the formula. Using Shannon expansions, we can express a non-read-once formula using a number of (exponential, in the worst case) read-once formulas and thereby compute its probability. Given a boolean formula  $\lambda$ , and a variable  $x$  that appears in  $\lambda$ , we can represent  $\lambda$  by means of the identity:

$$\lambda = (x \wedge \lambda_{x=1}) \oplus (\bar{x} \wedge \lambda_{x=0})$$

$\lambda_{x=1}$  and  $\lambda_{x=0}$  are called the Shannon co-factors of  $\lambda$  w.r.t.  $x$ .  $\lambda_{x=1}$  is the boolean formula obtained by setting  $x = 1$  in the formula and  $\lambda_{x=0}$  is the boolean formula obtained by setting  $x = 0$  in the formula.  $\bar{x}$  denotes the negation of variable  $x$ . The XOR between the terms is because the assignments satisfying the first term are mutually exclusive of the assignments satisfying the second term.

Consider the boolean formula  $\lambda = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4)$ . Suppose we want to expand the formula along  $x_2$ . The expansion we get is:

$$x_2(\lambda_{x_2=1}) \oplus \bar{x}_2(\lambda_{x_2=0}) = x_2 \wedge (x_1 \vee x_3) \oplus \bar{x}_2 \wedge (x_3 \wedge x_4).$$

Note that both  $(x_1 \vee x_3)$  and  $(x_3 \wedge x_4)$  are read-once and hence their probabilities can be evaluated easily. Also, since the two terms in the expansion are mutually exclusive, we can simply add their probabilities to get the probability of the original boolean formula.

## 2.2 Related Work

The broad field of probabilistic databases has seen a lot of work in recent years. In this section, we attempt to list the literature which is highly relevant to our work and contrast them to the contributions made by us. Owing to the inter-disciplinary nature of the research, the work described in the dissertation relates to various areas of computer science including machine learning and graph theory. Here, we classify the related work according to the various fields of research and within each field, we describe how this work relates to the prior work.

### 2.2.1 Probabilistic Databases

Efforts to represent uncertainty in relational databases have been around for a fairly long period of time, thereby generating a wide variety of techniques. Perhaps the earliest work in this field was by Barbara et al. [9], which explored *attribute uncertainty* models, where the value of an attribute can take multiple different values, each value weighted with a probability. The work by Fuhr and Rolleke [42] was one of the earliest works that proposed a simple model of a probabilistic database based on *tuple uncertainty*. They used it in order to incorporate information retrieval into database systems. Each tuple was assigned a *probability*, which specified the likelihood that it belongs to the database. Probview [70] was another early research project which proposed using a probability range instead of a single number (since an actual value for the probability itself may be uncertain). They develop query processing techniques based on Linear Programming. A renewed interest in the field of probabilistic databases began after Dalvi et al. [27] proposed a simple prob-

abilistic database model based on tuple uncertainty, and used *possible world semantics* [42], an intuitive semantics for query evaluation. Dalvi et al. [27] proved that general query evaluation over probabilistic databases, based on possible world semantics is  $\#P$ -complete and characterized the class of conjunctive queries that allow polynomially computable plans, which was termed as *safe plans*. In further work [26], they also prove the dichotomy of query evaluation, i.e., *hierarchical conjunctive queries* are polynomially computable, while the non-hierarchical queries are  $\#P$ -complete.

Following the work of Dalvi et al. [27], several other research literature, focusing on query evaluation in probabilistic databases [110, 95, 92, 5, 4, 96, 15] was developed, based on possible world semantics. One example is the Trio system at Stanford [110, 95] which proposed a two step process for query evaluation in probabilistic databases. In the first step, the system keeps track of the *lineage* of the output tuples as a boolean formula. In the second step, the system evaluates the probability of the lineage. We adopt the same strategy to evaluate conjunctive queries as we show in Chapter 6. Sen et al. [96] propose to represent arbitrary probabilistic databases using probabilistic graphical models [86]. They also propose query evaluation techniques based on *inference* algorithms in graphical models. We will discuss this work in detail in Section 2.1. Sen et al. [96] also showed the connection between safe plans and probabilistic graphical models by proving that safe plans correspond to tree structured PGMs, for which inference is known to be polynomially computable.

While this prior work has made great strides in our understanding of how to manage large-scale uncertain data and how to evaluate various types of queries on them, only a handful of these systems can handle correlated data effectively. Sen et al. [96, 97] and Antova et al. [5, 4] have addressed issues

in representing and querying complex correlations in probabilistic databases. However, their proposed techniques are not scalable to very large databases that we would like to handle in our applications. In our work, we extend the work of Sen et al. [96] by developing an index data structure called *INDSEP* that enables scalable query processing over large-scale correlated probabilistic databases.

Another related work is that of Bravo et al. [12], who address the problem of evaluating What-If queries (which they call as MPF queries) using relational database techniques. They represent each conditional probability distribution as a separate relation, and show how to scalably evaluate what-if queries using relational operators. Our approach towards evaluating what-if queries is largely complementary to their approach since we focus on directly evaluating what-if queries on the junction tree. They focus on a special kind of probabilistic database where the number of uncertain entities is small, but the conditional probability distributions (that quantify the correlations) are large. We address the complementary problem where the size of the database is very large, but the probability distributions encoding the correlations are relatively small.

Also related to our research is the literature on aggregate query processing over probabilistic streams. Jayram et al. [55] and Cormode et al. [22] present algorithms for computing expected values of aggregates such as MIN, MAX, AVG etc over probabilistic data streams. Our work differs from this in two aspects. First, our focus is on computing the exact probability distribution of the aggregates rather than just the expectations. Second, our techniques can handle the strong spatial and temporal correlations present in real-world probabilistic data streams, which this prior work ignores.

## 2.2.2 Inference in Graphical Models

As shown by Sen et al. [96], query evaluation in probabilistic databases is a specific instance of the problem of *inference* in probabilistic graphical models (PGMs). Therefore, much of the work on that topic in machine learning is highly related to our research. As discussed in Section 2.1, a probabilistic graphical model is a concise representation of a large joint probability distribution among many random variables that exploits the conditional independence relationships that exist in the data. The class of PGMs contain *Bayesian networks* [86] and *Markov networks* [24] as special cases. Bayesian networks only include directed dependencies and Markov networks only allow undirected dependencies among the random variables.

Efficiently evaluating inference queries over PGMs has been a major research area in the probabilistic reasoning community for many years. Pearl’s belief propagation algorithm [86] was one of the earliest algorithms that has been proposed for inference in graphical models. However, this algorithm was designed primarily for exact inference in tree structured graphical models [86]. In the presence of loops, there is no guarantee that the algorithm works correctly. A number of exact algorithms were developed following this work such as variable elimination [113], cutset conditioning [76] and junction trees [40]. Each of these algorithms proposed are exponential in the *treewidth* [94] of the graphical model. We discussed variable elimination and junction tree methods in detail in Section 2.1. Our work in the dissertation builds upon the junction tree algorithm. The *INDSEP* data structure which we propose in Chapter 5 improves the traditional junction tree inference algorithms by orders of magnitude. We expect our techniques to be useful in answering inference queries

over large-scale graphical models as well.

More recent work in machine learning has focused on a new class of PGMs called *first-order* graphical models (FO-models). FO-models are essentially PGMs with an additional layer of specification that uses first-order rules to specify correlations among classes of random variables. Essentially, the same correlation applies to all random variables belonging to the respective classes. This allows FO-models to be compactly represented. Examples include Probabilistic Relational Models [44] and Markov logic networks [93]. A new class of inference algorithms called *lifted inference* algorithms aims to exploit the symmetry provided by FO-models to achieve more efficient inference. The basic idea behind lifted inference is to develop inference algorithms that sum over *sets* of random variables and multiply sets of factors, instead of summing over each random variable and multiplying each of the factors individually. Poole [87] was the one of the first to show that variable elimination can be modified to directly work with FO-models to avoid grounding out the PGM during inference. Wang et al [108] developed the BayesStore probabilistic database system that exploits the use of FO-models proposed by Poole et al [87] to compactly store and query probabilistic databases. Subsequently, Braz et al. [30, 31] further developed on Poole’s work and introduced two techniques for lifted inference known as *inversion elimination* and *counting elimination*. Sen et al. [97] develop a general algorithm for lifted inference based on the graph theoretic concept of *bisimulation* [82, 63] to exploit shared correlations that frequently occurs in a probabilistic database.

### 2.2.3 Indexes for Probabilistic Databases

Indexes such as B-trees and R-trees have long been used in the database community for speeding up query evaluation in relational databases. In the probabilistic database research, there has been recent work on developing indexes for improving the efficiency of query processing. The earliest works in this area [17, 102, 101] develop indexing techniques based on R-trees and inverted indices for the efficient execution of *nearest neighbor* queries and *probabilistic threshold queries*. However, most of these approaches assume independence between different data tuples and do not work for the data in our application.

Perhaps the most closely related work to ours is the recent work on indexing *Markovian streams* by Letchner et al. [71]. The authors exploit the correlation structure exhibited by such streams to design a *Markov chain index*. In Chapter 5, we illustrate the similarities and differences between the Markov chain index and *INDSEP*. The concept of *shortcut potentials*, which we illustrate in Chapter 5 is loosely based on this idea. The authors show how to efficiently execute pattern identification queries over Markovian streams using this index. The proposed index structure however requires the Markovian property which significantly limits the types of correlations that can be handled. Our index structure, on the other hand, can handle arbitrary types of correlations (as long as inference is not intractable).

There has been work on indexing graphical models in the machine learning community as well. Recently Darwiche et al. [28] proposed a data structure called DTree that has superficial similarities to our approach. A DTree specifies a recursive decomposition of a PGM into its constituent probability functions,



and provides a recursive framework for executing inference queries. However the types of queries supported are limited to computing evidence probabilities (e.g.,  $p(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n)$ ). Also, DTrees are in-memory data structures and cannot handle large, disk-resident databases. Finally, a DTree is by definition a binary tree, which significantly restricts its performance benefits; specifically, we may still need to access the entire tree to answer an inference query (as the authors note, the key benefit of a DTree over a junction tree is its lower memory footprint and not necessarily lower querying times).

## 2.2.4 Sensitivity Analysis

### Sensitivity Analysis in Bayesian networks

Sensitivity Analysis problems in graphical models have been studied by Chan et al. [13] and Kjaerluff et al. [65]. Van der Gaag et al. [65] show that sensitivity of an output variable in a Bayesian network with respect to each input CPT parameter can be computed in  $O(n)$  time ( $n$  is the size of the network) using techniques similar to junction tree belief propagation. The number of CPT parameters is typically exponential in the size of the largest factor in the Bayesian network. Chan et al. [13] have continued this line of work to compute influences of *pairs* of input parameters. These analyses are more general than our methods since we make assumptions regarding (1) tree structured nature of the Bayesian network, (2) boolean input variables, implying much fewer input parameter and, (3) we are only interested in measuring the sensitivity of one designated child-less output node. Owing to the simplified nature of our problem, we have been able to provide more efficient algorithms in our case. Further, in our work we estimate influences of the input tuple probabilities over

Boolean formulas and complex aggregation functions, which are not considered in prior work.

### **Sensitivity Analysis in Probabilistic Databases**

Re et al. [92] originally considered the problem of computing influential input tuples and explanations for boolean conjunctive queries. They propose alternative forms of *approximate lineage* and develop algorithms to recover influential tuples from them, while our focus is on efficient exact computation based on the original lineage itself. Our proposed definitions of influence and explanations (in Chapter 8) generalize Re et al. (Section 8.1)’s definition (which is restricted to boolean conjunctive queries) and are applicable to a variety of queries. Our techniques also handle aggregation and top-k queries in addition to conjunctive queries.

### **Causality in Databases**

In recent work, Meliou et al. [77] develop the notion of causality of input tuples on the result tuples of a query, based on the fundamental notion of causality proposed by Halpern and Pearl [49, 50]. Informally, a tuple  $t$  is a cause for a query result if there exists a possible world in which the presence/absence of  $t$  changes the query result for that world. The *responsibility* of a tuple  $t$  on the query result, as defined in Meliou et al. [77] relates to the number of possible worlds that are affected by the presence/absence of the tuple. We describe the connection between responsibility as defined here and influence in Section 8.1.4.

# Chapter 3

## PrDB System Overview

In this chapter, we briefly provide an overview of the PrDB system. We start by describing the various components of the system. The main components of the PrDB system are

- Relational storage system, which stores the tuples and the associated correlations.
- INDSEP, which is an index built on top of the junction tree corresponding to the probabilistic database.
- Parser, that is used to insert new tuples into the probabilistic database.
- Query Processor, which is used to evaluate *extraction* queries, *inference* queries, *aggregation* queries and *lineage* queries over the probabilistic database.
- Probabilistic modeling system, which applies *dynamic probabilistic models* to uncertain data and subsequently generates correlated data.

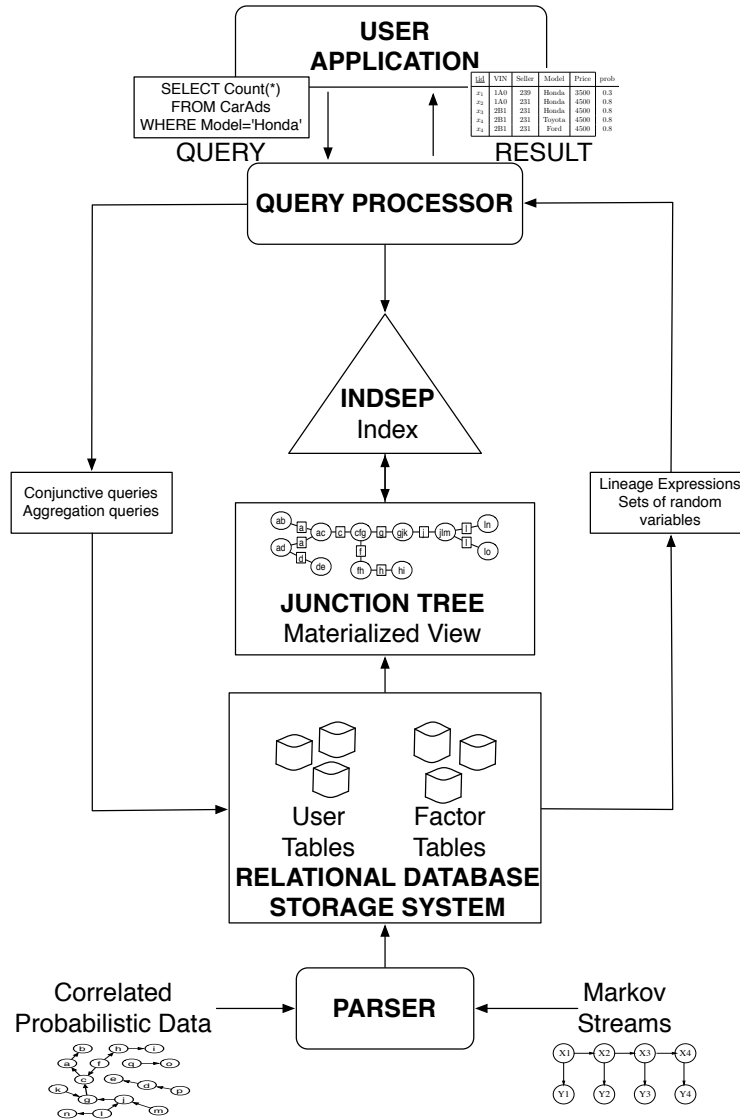


Figure 3.1: Various components of the PrDB system. Probabilistic data is inserted by the user via the parser. The correlations are stored in the factor tables in the system. We propose a junction tree-based materialized view representation of the database and index it using the *INDSEP* data structure. The query processor interacts with the *INDSEP* and the underlying user tables.

The different components of the system are schematically shown in Figure 3.1. We start by discussing the relational storage system. We illustrate the probabilistic modeling system in detail in Chapter 3.5.

### 3.1 Relational Storage System

We use a relational DBMS to store the tuples and the associated uncertainty. In this section, we describe the relational database schema that we use for this purpose. We also discuss the language that we have developed for the users to interact with the system. We illustrate the storage system with an example. Consider the probabilistic database shown in Figure 3.2(a). It has two relations  $S$  and  $T$  both of which have tuple uncertainty and attribute uncertainty. For instance, the tuple  $s_1$  in relation  $S$  has tuple uncertainty since it has a column marked ‘?’. Also, the attribute  $B$  is uncertain for both tuples. Further, there are correlations in the model, between tuples  $t_1$  and  $t_2$  in relation  $T$  and between attributes  $t_1.B$  and  $t_1.C$  in  $T$ . The correlations are shown as a graph in Figure 3.2(a). In order to store all of the uncertainty and the correlation information in our system, we have developed a relational schema with 5 relations as shown in Figure 3.2(b). We now explain the tables in the schema.

1. The *rvs* (random variables) table stores the list of random variables in the probabilistic database along with information corresponding to its domain. As we discussed in Chapter 2, there are 2 types of random variables in PrDB – the boolean random variables corresponding to tuple uncertainty and the random variables corresponding to attribute uncertainty. Each random variable name is assigned using the *key* attribute of the corresponding relation and the column attribute name. For example, the uncertain attribute corresponding to the tuple  $s_2$  in  $S$  is given the name  $s_1.B$ . Tuple uncertainty random variables are given the special name *EU* (exists uncertainty). Note that although we indicate the

uncertainty attribute using ‘?’ symbol, we use the column name “EU” in our system implementation. The table also stores the information corresponding to the domain of the random variable. For instance, the domain of a tuple uncertainty attribute is given by {true,false}. However, instead of storing the same domain for several variables separately which can be quite inefficient for large domain random variables, we normalize and store the actual domain in another relation *domains*. In *rvs*, we store the pointer to the actual domain using a domainId (*domID*) as a foreign key. Note that the domains of  $s_1.B$ ,  $s_2.B$  and  $t_1.B$  are all denoted as *carDomain*, which corresponds to the actual set {Honda, Toyota}. As we will describe later, these domIds can either be specified by the user or they can be assigned automatically by the system.

2. We store the information corresponding to the factors in the relations *sharedfactors*, *factors* and *factorrvs*. Again, to exploit the fact that multiple tuples can have the same factor associated with them, we store the actual numbers corresponding to the factors separately. In the shared factors relation, we store the names of all the factors/correlations in the probabilistic database. For instance, the factor  $f_{t_1.EU,t_2.EU}$ , which corresponds to the mutual exclusion correlation between tuples  $t_1$  and  $t_2$ , is stored here. Note that the actual mutex correlation is stored the *factors* relation. We store the actual factor numbers using a varchar attribute type. For instance, we store the mutual exclusion factor using the string “0 0 1; 0 1 1; 1 0 1; 1 1 0”. This is a string version of the tabular factor with four rows (separated by semicolons). The ordering of variables in a factor is stored in a separate relation called *factorrvs*.

We note here that the values in the above tables are not populated by the user. The user interacts with a system using a declarative language with useful constructs. We now describe the language that we have developed for this purpose.

## 3.2 Parser and Language

We start by describing the declarative syntax using which the users can insert uncertain data and specify the correlations in the data. We have developed constructs that allow the users to specify *shared* correlation structures, i.e., users can assign correlations associated with multiple tuples using a single input statement provided they share the same correlation. We illustrate the language and the schema of the internal database with simple examples. The parser allows users to define correlations as factor objects and also insert them against corresponding tuples in the database.

Further, traditional SQL insert statements for inserting tuples into tables and the create table statements are also supported. The complete list of language constructs we support is as follows.

```
[1] DEFINE DOMAIN <dom_name> (<v1>, <v2>, ..., <vn>)
[2] INSERT DOMAIN (<dom_name> | (<v1>, <v2>, ..., <vn>)) IN <table> ON <attr>
[WHERE <predicate>]
[3] DEFINE FACTOR <function_name> (...;...;...)
[4] INSERT FACTOR (<function_name> | (...;...;...)) IN <tables> ON <rvlist>
[5] INSERT FACTOR (<function_name> | (...;...;...)) IN <tables> ON <attr_list>
[WHERE <predicate>]
```

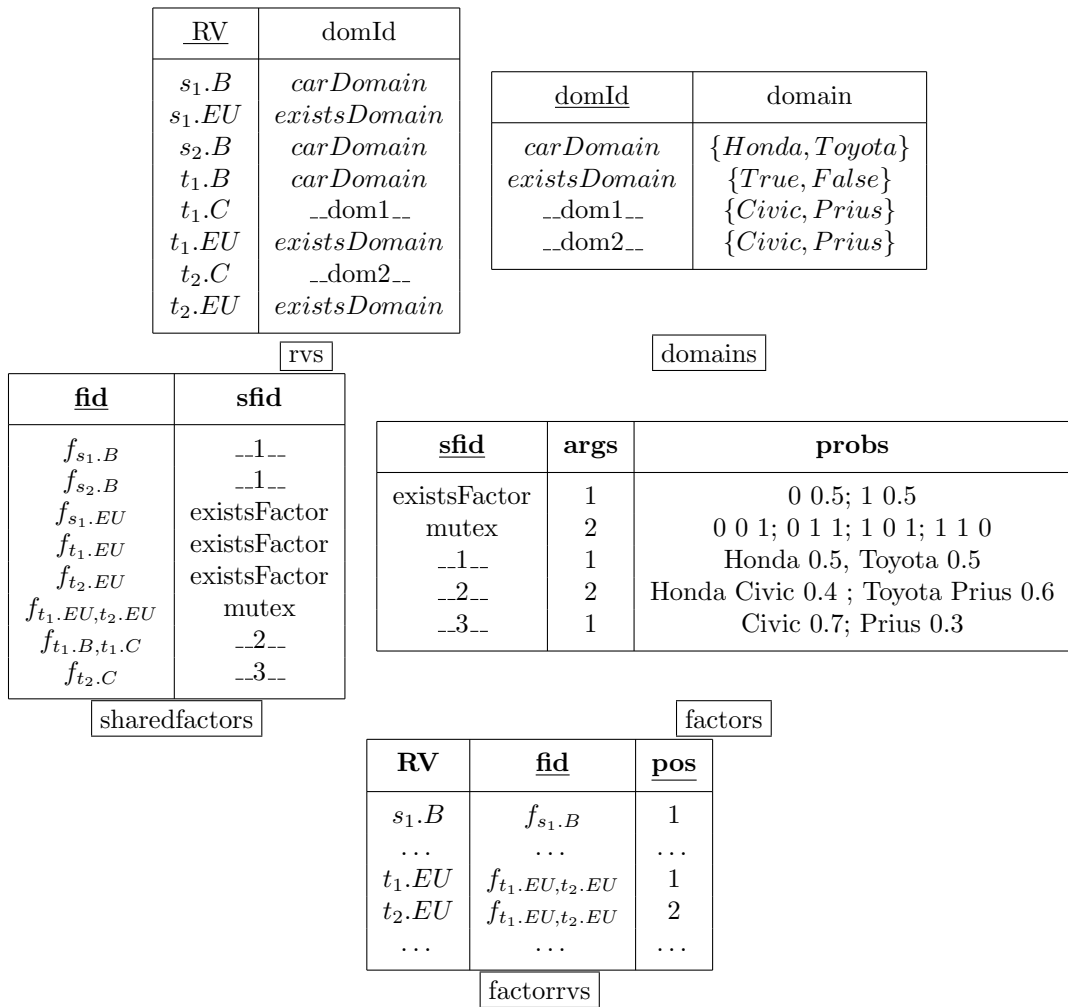
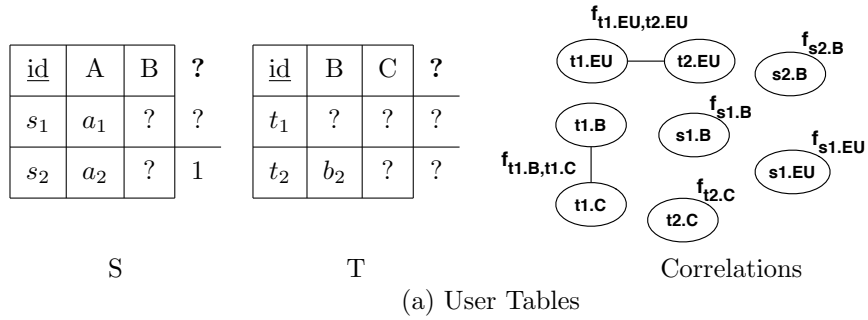


Figure 3.2: Schema of the PrDB model. User tables are shown in (a). Internal system tables are shown in (b)



Construct/Example	Meaning
<u>define domain carDomain</u> <u>(“Honda”, “Toyota”)</u>	This statement defines a new domain given by {“Honda”, “Toyota”}, which can now be assigned to the relevant random variables.
<u>insert domain carDomain</u> <u>in S on B</u>	This statement assigns the domain carDomain to the random variables $s_1.B$ , $s_2.B$ . Note that we can exploit common domains between random variables by using only statement. Note that since we explicitly specify the name of the domain to be inserted, the name “carDomain” is inserted in the RV table. However, if we had specified the domain directly, an arbitrary name is automatically assigned by the system.
<u>define factor mutex (0 0</u> <u>0.4, 0 1 0.3, 1 0 0.3, 1 1 0)</u>	This statement defines a new factor. The example shown here defines a mutual exclusion factor with the given probabilities, which can now be inserted to the corresponding tuples.
<u>insert factor mutex in T</u> <u>on <math>t_1.EU</math>, <math>t_2.EU</math></u>	This statement is used to assign factors to the relevant tuples. The example shown here assigns a factor on tuples $t_1$ and $t_2$ .
<u>insert factor (“Honda”</u> <u>“Civic” 0.4 ; “Toyota”</u> <u>“Prius” 0.6) IN T ON B,C</u> <u>WHERE <math>id = t_1</math></u>	We can also perform the above statement on multiple tuples by including a <where> predicate. This allows us to specify shared factors on to multiple tuples very easily.

Figure 3.3: Various constructs to insert data and correlations in PrDB

[6] INSERT VALUES (...)

[7] CREATE TABLE (...)

The list of statements to generate the database of Figure 3.2 is shown in Figure 3.4.

```

create table R (id varchar(20), A varchar(100), B varchar(100), EU char) ;
create table S (id varchar(20), B varchar(100), C varchar(100), EU char) ;
define domain existsDomain (1, 0) ;
define domain carDomain (Honda, Toyota) ;
insert into S values("s1", "a1", ?, ?) ;
insert into S values("s1", "a1", ?, ?) ;
insert into T values("t1", ?, ?, ?) ;
insert into T values("t2", "b2", ?, ?) ;
insert domain existsDomain IN T on EU ;
insert domain existsDomain IN S on s1.EU ;
insert domain {"Civic", "Prius"} IN T on t1.C ;
insert domain {"Civic", "Prius"} IN T on t2.C ;
define factor existsFactor (0 0.5, 1 0.5);
define factor mutex (0 0 1; 0 1 1; 1 0 1; 1 1 0);
insert factor ("Honda" 0.5, "Toyota" 0.5) on B ;
insert factor existsFactor in S on s1.EU ;
insert factor existsFactor on T.EU ;
insert factor mutex IN T ON t1.EU,t2.EU ;
insert factor ("Honda" "Civic" 0.4 ; "Toyota" "Prius" 0.6) IN T ON B,C
WHERE id = t1 ;
insert factor ("Civic" 0.7; "Prius" 0.3) in T on t2.C ;

```

Figure 3.4: List of statements to generate the database of Figure 3.2

### 3.2.1 Parser Implementation

The `define domain` statement requires us to insert a new domain tuple in the *domain* table. We use the name given by the user as the `domID` for this domain. The `insert domain` statement requires us to insert tuples into the *rvs* table. Note that the statement can either specify an already defined domain, which is inserted into the table, or indicate a new set in which case, the parser assigns a new name to the domain. The more interesting statement is the *insert factor* statement which requires us to insert tuples into several tables. Consider the statement `insert factor ("Honda" "Civic" 0.4 ; "Toyota" "Prius" 0.6) IN T ON B,C WHERE id = t1`. To implement this, we first run the SQL query with the given predicate on the given relation *T*

and compute the list of output tuples. We assign the factor to each output tuple's B and C attributes. Just as before, to insert the factor, we need to insert into both the *factors* and the *sharedfactors* tables.

### 3.2.2 Factor Semantics

Now, we discuss the semantics of the inserted factors. Consider a tuple  $t$ . There are two sets of factors that may get associated with the tuple.

1. Tuple-specific factors defined for this tuple alone using [4].
2. Shared factors defined over portions of the relations satisfying the **WHERE** predicate using [5].

In general the factors supplied by the users may not be *consistent*. We illustrate this with an example. For instance, suppose that the user provides the following statements.

```
insert factor ("Honda" 0.5; "Toyota" 0.5) in S on B where id = s1  
insert factor ("Honda" 0.7; "Toyota" 0.3) in S on s1.B
```

Although neither of the statements are problematic by themselves, each statement assigns a different prior factor on the variable  $s_1.B$ . More severe inconsistencies can occur such as: (1) a mutex factor on two tuples  $t_1$  and  $t_2$  (2) and a perfect correlation factor between  $t_1$  and  $t_2$ . A couple of workarounds for this include ignoring the second statement since the first statement, which is already parsed is contrary to the second statement. Another option is to simply multiply the factors together without examining the contents of the factor. In our system, we take the approach of multiplying the factors provided by the user. However, we take care to ensure that the multiplication operation does not result in zero valued factors.

### 3.3 Junction tree & INDSEP

After the user specifies the probabilistic database, the system builds the PGM corresponding to the user specified correlations. The PGM is constructed by extracting all the factors from the disk. We do this by first performing a join between the *factors* and the *sharedfactors* table (equating the *sfid* column). Next, for each of the tuples in the resulting relation, we look for the corresponding tuples in the *factorrvs* relation and determine the ordering of the variables in the factor. Using all of this information, we can construct all the factors in the PGM. After constructing the PGM corresponding to the database, we build the junction tree of the PGM using the algorithm shown in Chapter 2. Following this, we build our index structure, *INDSEP* over the junction tree. We will discuss INDSEP in detail in Chapter 5. Note that since the user is allowed to specify correlations over arbitrary sets of random variables, the marginals defined by the different correlations need not agree and are hence ambiguous. In this work, we simply assume that such ambiguities do not arise. Both the junction tree and its corresponding INDSEP are currently implemented as in-memory objects.

### 3.4 Query Processor

The system currently supports three kinds of queries:

1. Extraction Queries: These are specified using a set of random variables.

An example of an extraction query was shown in the Introduction (Section 1.2.2).

2. Inference Queries: These are specified by a set of random variables. For

instance,  $\{t_1.EU, t_2.EU\}$  specifies an inference query and requires us to evaluate the probability distribution  $p(t_1.EU, t_2.EU)$ .

3. Aggregation Queries: These are specified using SQL queries over the probabilistic database (user tables). An example of such a query was shown in the Introduction (Section 1.2.2).
4. Conjunctive/Lineage Queries: These are specified using Select-Project-Join SQL queries over the probabilistic database (user tables). An example of such a query was shown in the Introduction (Section 1.2.2).

Inference queries and Extraction queries can be directly evaluated over the junction tree since they are specified in terms of random variables. However, aggregation queries and conjunctive queries are specified using SQL queries over the probabilistic database. They need to be translated to queries over the random variables. Hence, the query processor employs a two step process for computing the output results. For aggregation queries, the query processor first determines the set of random variables over which it needs to aggregate. In addition, it also needs to determine if any of the tuples being aggregated also exhibit tuple uncertainty, since this influences the value of aggregates like `AVERAGE` and `SUM`. In the second step, the value of the aggregate is actually computed. We will discuss the algorithms for evaluating aggregates in Chapter 5.

For conjunctive queries, the query processor first determines the output tuples of the query evaluation and simultaneously keeps track of the lineages of the output tuples. In the second step, the probability of the lineage formula is evaluated on the junction tree. This step is discussed in detail in Chapter 6. The lineages of the output tuples are constructed using a query rewrite

procedure as shown below.

We use a simple query rewrite to track the lineages of the result tuples of a conjunctive query by exploiting the `concat` and the `group_concat` constructs of SQL. Given an SPJ query such as, SELECT <S> FROM <F> WHERE <W>. We rewrite this query as:

<pre> SELECT &lt;S&gt;, GROUP_CONCAT (SEPARATOR '+' ) AS EU FROM {   SELECT &lt;S&gt;, CONCAT(t1.EU, '*', t2.EU, '*', .. AS EU   FROM &lt;F&gt; WHERE &lt;W&gt; } GROUP BY &lt;S&gt; </pre>
---

Here,  $t_i$ 's are the tables that are contained in the FROM clause of the input query. We also assume that each of the input tables has an attribute named "EU" that represents the random variable corresponding to its tuple uncertainty. We note here that this technique is designed for tuple uncertainty probabilistic databases. To handle an attribute uncertainty probabilistic database, we can convert it into a tuple uncertainty probabilistic database as described in Section 1.1.2 and then use the same rewriting routine.

### 3.5 Probabilistic Modeling System

In this section, we provide details of the probabilistic modeling system. The input to the system is an uncertain data stream and the user specified probabilistic model (DPM) to be applied over the data. The output of the system is a model-based-view [35] that represents the correlated probabilistic data generated by the modeling process. We built the probabilistic modeling system using Java, and we use the open source Apache Derby (Java embedded

database system) [6] 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 4.3) in memory for efficient access; the particles are written to the database in background. We start by describing how to create DPM views from uncertain data in Section 3.5.1.

### 3.5.1 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 *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 3.5 shows an example of such configuration files for the HMM presented in Section 2.1.1. The configuration

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

Figure 3.5: (i) Conventions used in specifying the DPM; (ii) Configuration file for HMM-based view in Figure 2.1(i)

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 2.1.1.3) 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 3.5(i) shows the distributions we currently support. For example,  $N(\mu, \sigma)$  represents a normal distribution with mean  $\mu$  and standard deviation  $\sigma$ . The CPD for node 2 in Figure 3.5(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 3.5(ii) has a discrete distribution that was specified using a transition probability matrix in Figure 2.1(iii).
- *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 the pre-defined API shown below.

- **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.

Finally, instead of specifying the parameters explicitly using the configuration file or the API, the user may instead specify a training dataset from which to learn the parameters (*line 3* in the view creation syntax).

## Chapter 4

# Probabilistic Modeling of Uncertain Data

Dynamic probabilistic models are widely used in practice to model and to reason about complex real-world stochastic processes [56, 80, 79]. The simplest and most widely used examples of DPMs are *hidden Markov models (HMMs)* and *linear dynamical systems* (better known as Kalman filter models (KFM)). In this chapter, we illustrate how to apply *generic* DPMs to uncertain data and uncertain data streams. We do so, via the abstraction of a *model-based-view*, as shown in Section 4.1. Subsequently, we discuss how to efficiently represent model-based-views in a relational database and how to evaluate simple queries over them. We discuss how to keep the views up-to-date in response to updates to the input data stream in Section 4.3. We use an MCMC technique based on *particle filters* for view maintenance. Finally, we conclude with experiments in Section 4.4.







## 4.1 DPMs as Database Views

The abstraction of *model-based view*, proposed in [35], 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 [35]. Here, we extend this abstraction by allowing views to be defined using DPMs instead. Figure 4.1(i) shows the schema of the view that could be presented to the user with the BBQ DPM model (Figure 4.2). We briefly explain the BBQ DPM before providing details about the DPM-based view.

**BBQ DPM [34]:** Figure 4.2 depicts the DPM that we use as a running example in this chapter. 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  $M_0, \dots, M_t$ , 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).

As we can see from Figure 4.1(i), 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 (across 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

t	temp $T_t$	humid $H_t$
$\vdots$	$\vdots$	$\vdots$
3		
4		
5		
$\vdots$	$\vdots$	$\vdots$

SID	t	$T_t$	$H_t$	weight
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
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
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(i) DPM-based view      (ii) Associated Particle Table Architecture

Figure 4.1: (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)

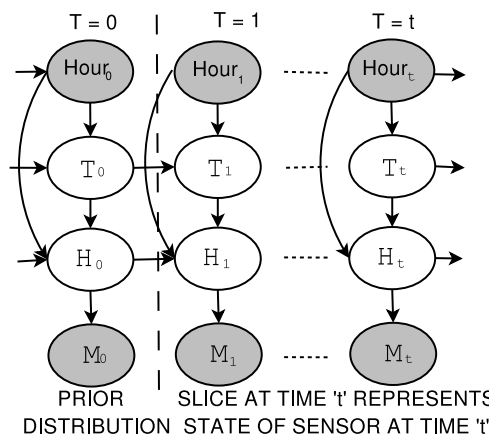


Figure 4.2: Graphical representation of the BBQ DPM used for modeling Intel Lab data (Section 6.4)

HMM-based view presented to the user in the fault detection example (Section 2.1.1.1)).

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). 4.1 (i), the *temperature* attribute is not known with

certainty The issue of querying and representing such probabilistic data has received much attention in recent years [9, 70, 16, 110, 27, 3, 96], 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). example, in Figure 4.1 (i), *humidity* and the *temperature* attributes are correlated.
- **inter-tuple correlations:** that exist between attributes of different tuples (e.g.,  $T_t$  and  $T_{t+1}$ ). 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 Chapter 5, we 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 occurrence 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 4.3). The set of particles represents the joint distribution over the attributes in the view. Figure 4.1(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 4.1(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  $1/N$  [37].

## 4.2 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 3.5.1).
2. If the user specifies that the CPDs are to be learned using training data, an MLE-based *learning module* (see [59]) is invoked over the training data.
3. A particle table is created and initialized using the prior distributions (Section 4.3).
4. The particle table is continuously updated by the *Update Manager* in response to the incoming data stream measurements (Section 4.3).

## 4.3 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* [37] 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 4.1, 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 4.2). Pseudocodes for these routines and a more comprehensive illustration is presented in [59].

**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  $i^{\text{th}}$  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  $p(T_{t+1}|T_t, \text{hour}_{t+1})$  and  $p(H_{t+1}|H_t, \text{hour}_{t+1})$  where  $\text{hour}_{t+1}$  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 at most  $L$  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.

## 4.4 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 [37].

#### 4.4.1 Experimental setup

##### **Dataset I: Moving Objects Dataset**

Moving objects databases have received much attention in recent years [89, 105, 16]. 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 2.1(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 [34, 83, 16] on managing noisy and incom-

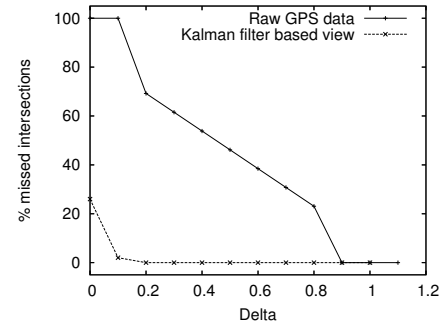
```

(a) SELECT u.OID, v.OID, u.time
FROM kfview u, kfview v
WHERE (u.time = v.time)
AND (|u. $\mu(x)$  - v. $\mu(x)$ | <  $\delta$ )
AND (|u. $\mu(y)$  - v. $\mu(y)$ | <  $\delta$ )

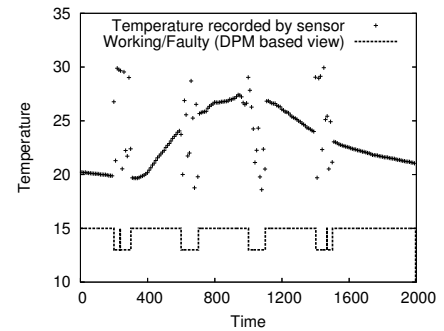
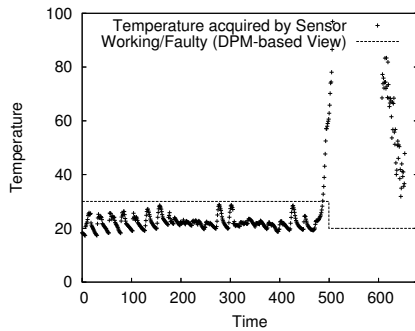
(b) SELECT kfview.x, kfview.y
FROM kfview
WHERE kfview.OID = 4

```

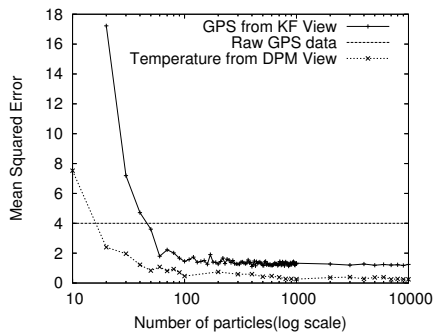
(i) (a) Intersection query (b) Trajectory query



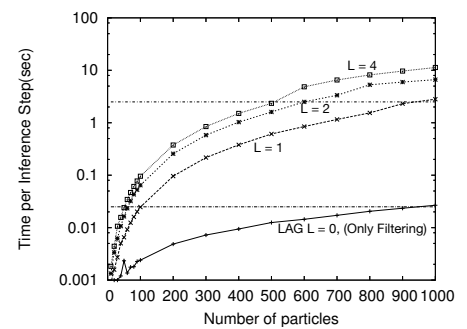
(ii) *kfview* captures all intersections



(iii,iv) Faulty readings removed by HMM-based view.



(v) Accuracy of Inference



(vi) Smoothing performance

Figure 4.3: (i) Queries used in Experiments; (ii) % of missed intersections as a function of  $\delta$  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).

plete 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 [74] that consists of traces from a 54-node sensor network deploy-

ment 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 [34] in power-aware sensor networks, where acquiring all attributes is expensive. We run a series of processing tasks over this data.

**Step 1: Remove Incorrect Data** Detect failure times of sensor nodes using an HMM-based view (Figure 2.1(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 4.2).

**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.

## 4.4.2 Experimental Results

### 1. Applying DPMs to data is critical

**Dataset I:** The *intersection* query in Figure 4.3(i) measures the number of times at which two particles are closer than a specified distance  $\delta$ . 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 4.3(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  $\delta$ . *kfview* on the other hand, is able to capture most of the real intersections.

**Dataset II:** Figures 4.3(iii),(iv) show the results of executing Step 1, i.e., detecting the failure times for sensors. As we can see from Figure 4.3(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 4.3(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 4.3(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 4.3(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.

### 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,  $L = 0, 1, 2, 4$ .

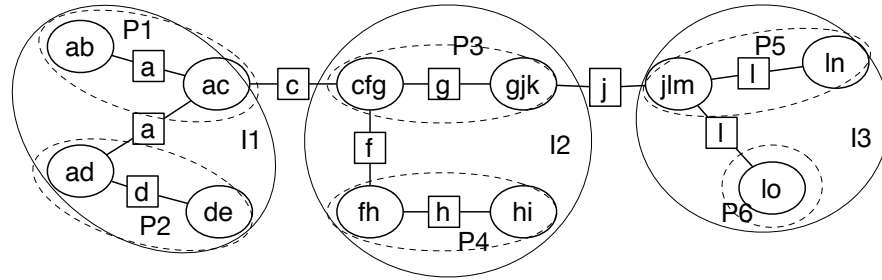
The results obtained are shown in Figure 4.3(vi). We find that the execution time increases linearly with increase in the number of particles (as y-axis is in log scale, this cannot be explicitly seen). If we perform only filtering, the inference time is very small; we process more than 1000 particles 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 4.3(v), this may be enough to achieve sufficient accuracy. We are considering “lazy” smoothing strategies where we perform smoothing occasionally (not at every time step) and only when it is essential.



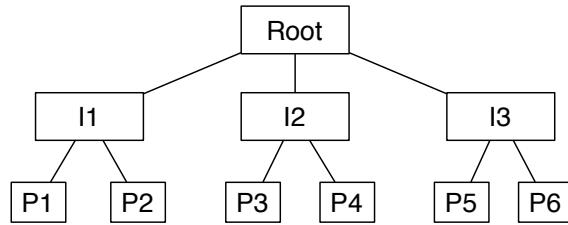
# Chapter 5

## INDSEP

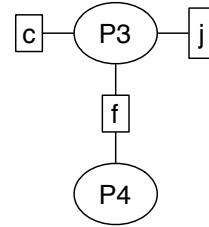
In this chapter we introduce our data structure called *INDSEP* for junction trees that enables scalable query processing over junction trees. *INDSEP* is a hierarchical tree based data structure that is based on a hierarchical partitioning of the junction tree. Its benefits are two fold. Not only does it function as an index in a database context: i.e., it enables selectively reading only relevant disk blocks into the memory for answering the required query, it also enables reduction in the query processing time itself, by orders of magnitude by using *shortcut* potentials. The outline of the chapter is as follows. We first provide an overview for shortcut potentials and illustrate how they can be used to improve the efficiency of query processing in Section 5.1.2. We then formally define the INDSEP data structure and provide an example for it, based on the junction tree of Figure 2.2(c). Next, we describe how to construct *INDSEP* using our hierarchical tree partitioning algorithm in Section 5.2. Next, we describe algorithms for evaluating *inference* queries and *aggregation* queries scalably over INDSEP in Section 5.3. Finally, we illustrate the mechanisms for keeping INDSEP up-to-date in response to updates to the



(a) Hierarchical Partitioning of the junction tree



(b) Corresponding index data structure



(c) Induced child tree stored in  $I_2$

Figure 5.1: (a) shows a hierarchical partition of the junction tree shown in Figure 2.2(c). Note that the separator nodes separating two partitions are replicated in both the partitions. The corresponding *INDSEP* data structure is shown in (b). The contents of the index node  $I_2$  is shown in part(c).

database in Section 5.4. We postpone the algorithms for processing lineage queries to Chapter 6.

## 5.1 INDSEP Data Structure

In this section, we describe our *INDSEP* data structure for indexing the junction tree that represents a probabilistic database. To build the *INDSEP* data structure, we hierarchically partition the junction tree into connected subtrees and subsequently construct the index. Before discussing the exact

algorithm for doing this, we specify the information stored in the different nodes of our INDSEP data structure.

### 5.1.1 Overview of the INDSEP Structure

At a high level, *INDSEP* is a hierarchical data structure that is built on top of the junction tree. Each index node in *INDSEP* corresponds to a connected subtree of the junction tree. Suppose we hierarchically partition the junction tree of our running example in Figure 2.2(c) as shown in Figure 5.1(a). Here, we first split the tree into three parts denoted  $I_1$ ,  $I_2$  and  $I_3$  (partitions are shown using large circles). After this, each part is further subdivided into smaller partitions. For instance,  $I_1$  is partitioned into parts  $P_1$  and  $P_2$  as shown in the figure with oval boundaries. The INDSEP data structure for such a hierarchical partitioning is shown in Figure 5.1(b). Here, the node  $I_2$  corresponds to the subtree spanning the cliques  $cfg$ ,  $gjk$ ,  $fh$  and  $hi$  along with the separator nodes  $c$  and  $j$ . Similarly, the node  $P_5$  corresponds to the subtree spanning the cliques  $jlm$  and  $ln$  along with separator nodes  $j$  and  $l$ . Note that separator nodes joining two partitions together are included in both the partitions. At a high level, each node in the data structure stores the following information about the subtree that it represents.

- **(C1)** Set of variables of PGM that are present in the subtree below this node. For example,  $I_2$  would store the set  $\{c, f, g, h, i, j, k\}$ . We note here that we are storing the *random variables that are part of the PGM* and not the clique identifiers of the junction tree.
- **(C2)** Pointers to index nodes of the children and parent pointers for index traversal.

- **(C3)** The set of separator potentials that join the children together. The set stored in  $I_2$  is  $\{p(c), p(f), p(j)\}$ .
- **(C4)** The graph induced on its children. Note that the separators that connect the children are also stored in this graph. The graph stored in  $I_2$  is shown in Figure 5.1(c).
- **(C5)** Set of *shortcut potentials* corresponding to the children of this node. We describe shortcut potentials in Section 5.1.2.

We describe each of the constituent components in more detail in turn.

**(C1)** Each node in the index structure needs to store the list of variables present in the subtree of each of the children of the node. The naive method of storing the list of elements of the set or even storing them as a bitmap is not feasible; if we had 1 million variables in the PGM, then each index node would occupy at least 125KB ( $> 30$  disk blocks) of space just to store the variables, which is a huge overhead. Instead, we store the set of variables under each child node using two data structures - a *range*  $[min, max]$  and an *addList*, i.e., the node contains all the random variables whose ids are either within the range  $[min, max]$  or if it is present in the addList. This *contiguous variable name* property is the key idea in reducing the amount of space taken by our index structure. We achieve this property in the index using a variable renaming step, which we illustrate while describing the index construction algorithm. In fact, we also preserve this property even while updates occur to the database, i.e., when new random variables are added to the database.

**(C2)** A node stores the pointers to the disk blocks that contain the child nodes of that node. Since a child node could either be another index node or a leaf, we also store the type of the child along with its pointer. In Figure 5.1(b),

the root node stores pointers to index nodes  $I_1$ ,  $I_2$  and  $I_3$ . Similarly,  $I_2$  stores pointers to the disk blocks containing  $P_3$  and  $P_4$ . A node also stores a pointer to its parent node.

**(C3)** A node stores the joint distributions of all the separators that are connected to its child nodes. This includes both the separators that separate the children of the node from each other, and the separators that separate a child node from a child node of the node's sibling. For instance, the node  $I_2$  stores the set  $\{p(c), p(f), p(j)\}$ .

**(C4)** In order to be able to perform path computation on the junction tree, we need to store, in each node, the graph induced on the child nodes. Since we are partitioning trees into connected subtrees, the induced graph is also singly connected. For simplicity, we also store the separator cliques that separate the child nodes from each other. The node  $I_2$  in Figure 5.1(b) stores the induced tree shown in Figure 5.1(c). As shown in the figure, each child subtree is treated as a virtual node and then the edges are determined between the virtual nodes and the separator nodes, i.e.,  $P_3$  and  $P_4$  are treated as virtual nodes and they are connected via the separator node  $f$ . A path between  $i \in P_3$  and  $k \in P_4$  should necessarily pass through  $f$ .

### 5.1.2 Shortcut Potentials

In this section, we describe shortcut potentials, a novel caching mechanism which we have developed, that can provide orders of magnitude reduction in query time. Consider the graph shown in Figure 5.2, which represents a path connecting the variables  $X$  and  $Y$  in a junction tree. As shown earlier, we can compute  $p(X,Y)$  using the following sequence of messages from the clique  $C_1$

towards  $C_3$ .

$$\begin{aligned}
 m_{12}(C, X) &= \sum_{A,B} p(A, B, C, X) \\
 m_{23}(D, X) &= \sum_{C,E,F} p(C, D, E, F) m_{12}(C, X) \\
 p(X, Y) &= \sum_D m_{23}(D, X) p(D, Y)
 \end{aligned}$$

However, there is some unwanted computation going on above, which can be avoided. For instance, the variables  $E, F$  in the above equations are merely summed out in the message  $m_{23}$  and are not required to pass information about the interesting variable  $X$  to  $C_3$ . Since the size of the probability distributions are exponential in the number of the operands, the presence of these unnecessary variables can lead to increase in query processing times. Instead, if we had the joint distribution  $p(C, D)$  stored in the above example, the computation can be faster: we would replace the computation of the message  $m_{23}$  with

$$m_{23}(D, X) = \sum_C p(C, D) m_{12}(C, X)$$

$p(C, D)$  worked well for this example because it was the joint distribution of all the separators of the clique  $C_2$ , i.e., it had enough information to *shortcut* the clique  $C_2$  completely. While the above example was trivial and the savings quite minimal, we realize the full power of these caches by introducing the notion of *shortcut potentials*. We define the following notion of a shortcut potential that would be beneficial for our purposes.

**Shortcut Potential:**

The shortcut potential for a node  $I$  in the index data structure is defined as the joint distribution of all the separator nodes that are *adjacent* to the node  $I$ . A shortcut potential for  $I$  allows us to short cut the subtree represented by

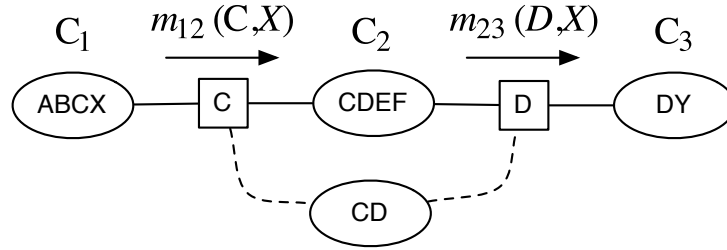


Figure 5.2: Illustrating overlays and shortcut potentials. Using the cached potential  $p(C, D)$  allows us to shortcut the clique  $C_2$  completely.

$I$  completely.

For example, in Figure 5.1(b), the shortcut potential for the node  $I_2$  is given by the joint distribution  $p(c, j)$ . Similarly, the shortcut potential for the partition  $P_3$  is the joint distribution  $p(c, j, f)$ . The separator nodes adjacent to a node are exactly the leaves in the induced child tree stored in the node. The size of a shortcut potential is the product of the domains of all the variables belonging to the set. Every index node stores the shortcut potentials for all of its children. Node  $I_2$  stores the shortcut potentials for  $P_3$  ( $p(c, f, j)$ ) and  $P_4$  ( $p(f)$ ). Note that storing the shortcut potentials for a node in its parent allows us to avoid accessing the node when the shortcut potentials have enough information. Whenever the size of a shortcut potential is larger than the size allotted for the index node (which is 1 disk block), we resort to approximating the shortcut potential, described next.

### Approximate Shortcut Potentials:

We provide 3 levels of approximations of shortcut potentials. Suppose the separators of node  $I$  are  $\{s_1, s_2, \dots, s_k\}$ . In the approximation scheme  $A_2$  we store the joint distributions of every pair of separators, i.e.,  $\bigcup_{i \neq j} p(s_i \cup s_j)$  (for a total of  $\binom{k}{2}$  joint distributions). In the approximation scheme  $A_3$  we store

the set of joint distributions of every triple of separators. When the exact shortcut potential is larger than the block size, we try to use scheme  $A_3$ ; when  $A_3$  also exceeds the block size, we resort to  $A_2$ . When  $A_2$  also exceeds the block size, we store a random subset of pairwise separators (scheme  $A_1$ ). For example, in partition  $P_3$ , we may choose to maintain the set  $\{p(c, j), p(j, f), p(c, f)\}$  if joint distribution over all the three variables is larger than the block size. We note here that using the approximation scheme  $A_3$  will only enable us to shortcut the subtree for 3 variable queries and  $A_2$  will enable us to shortcut the subtree only for 2 variable queries. In Section 5.4, we describe how to update shortcut potentials efficiently when updates occur to the database.

## 5.2 Index Construction

We now describe the steps involved in constructing the *INDSEP* data structure, given a junction tree and a target disk block size (each *INDSEP* node must fit in one disk block).

### 5.2.1 Hierarchical Partitioning

Our first step is to partition the junction tree into subtrees, each of which are smaller than the size of a disk block. We first assign a weight to each clique and each separator in the junction tree as the product of the domains of its constituent variables (i.e., the size of its joint distribution). The size of a partition is given by the sum of the sizes of the cliques and separator nodes that are present in the partition. Our objective function is to find the fewest number of partitions such that each partition can fit in a disk block (i.e., the space required to store the joint probability distributions corresponding to the



partition is less than the size of the disk block). This problem is identical to the tree partitioning problem considered in Kundu et al. [69].

We directly use the linear algorithm presented in their paper for constructing the partitions. At a high level, the algorithm first performs a depth first search on the tree and assigns level numbers to each node. After this, the algorithm iterates through the nodes starting from the lowest level (highest level number) of the tree and each node computes the weight of the subtree below itself. Once the weight of some node  $u$  exceeds the block size, we start removing the children below this node (children with highest subtree weight are removed first) and create a new partition for each of them, subsequently reducing the subtree weight of  $u$ . The algorithm continues until we reach the root. Kundu et al. [69] prove that the number of partitions generated using this algorithm is minimum.

After partitioning the junction tree, we treat each partition created as a virtual node and construct an overlay graph that is created on the virtual nodes. We also add the separator nodes that connect the partitions with each other to the overlay graph, and it is weighted as before. Each virtual node is weighted with the sum of the size of its shortcut potential (See Section 5.1.2) and the set of separator potentials that belong to it. At this point, we approximate the shortcut potential with approximation schemes  $A_1$ ,  $A_2$  or  $A_3$  if necessary.

We now perform Kundu's tree partitioning algorithm again on the overlay tree and recursively continue this process until we are left with exactly 1 virtual node, at which point, we create the root index node and complete the hierarchical partitioning. During the construction of the new partitions and index nodes, we also remember the disk blocks in which they were written

and fill out the parent and child block pointers for each node in the data structure accordingly.

## 5.2.2 Variable Renaming

We perform a *variable renaming* step after the hierarchical partitioning step in order to achieve the contiguous variable name property described earlier. We sort the leaves of the index tree (which correspond to tree partitions) in an *in-order* fashion and assign ids to the variables in the leftmost partition and proceed further to the next partition. Starting from 0, whenever we identify an unassigned variable, we give it an id equal to 1 higher than the previously assigned variable. After this step, each partition contains variables that are either contained in a closed interval  $[min, max]$  or belong to the set of previously numbered variables, which we store in the *addList*. The variables in the *addList* are exactly equal the set of variables in the separator that connected the previous partition with this. (The proof for this is quite trivial: Each partition is assigned a sequence of ids from min to max for the newly seen variables in the partition, the already existing variables will be in the *addList*, these are exactly the ones in the separator. The *running intersection property* of the junction tree guarantees this.) The number of variables in the *addList* are therefore much smaller when compared to the clique sizes. By performing *variable renaming*, we have effectively reduced the space consumed by the index node from 125 kB (for storing 1 million variables, see Section 5.1.1) to just a few bytes. We note here that we store the mapping between the old variable names and the new names in another relation, which may be indexed using B+-trees or hash indexes.

### 5.2.3 Assigning Range Lists and Add Lists

After each leaf of the index data structure is assigned the range lists and the add list, we recursively update the index. For each internal node in the index data structure, we assign its range list by merging the range lists of its children. Also, we scan the addLists of the child nodes and include the nodes which do not belong to the range of the node in its addList. In addition, we assign the shortcut potentials of child nodes to the current node. We continue this recursion till we reach the root, at which point all the index nodes have been updated.

Note that once the index is constructed using the above approach, it is guaranteed to be *balanced*, owing to the bottom-up nature of the algorithm. However, when updates occur to the database, it is difficult to guarantee that the index remains balanced. We currently propose to periodically reorganize the index to keep it balanced.

## 5.3 Query Processing

In this section, we provide algorithms for executing inference queries, aggregation queries and extraction queries over a probabilistic database by exploiting the *INDSEP* data structure.

### 5.3.1 Inference/Extraction Queries

As illustrated earlier, inference queries can be solved by constructing a tree joining all the query variables and then running the HUGIN algorithm over it. We use our INDSEP data structure to determine a small tree joining the

query variables by exploiting the relevant shortcut potentials that are present in INDSEP, i.e., we replace large sections of the trees with shortcut potentials whenever possible.

Our query processing algorithm is shown in Algorithm 1. It is a recursive algorithm on the *INDSEP* data structure. We first access the root block of the index and search for the query variables in the separator potentials. If they are not all present here, then we look for the query variables in the child nodes by making use of the range lists and the addLists present in the root. At this step, each query variable is assigned to a child node of the root. We mark each of these child nodes as Steiner nodes and compute the smallest Steiner tree  $S$  connecting the Steiner nodes in the induced child tree of the root node. Now we recurse along each node of the Steiner tree, and concatenate their outputs together to compute the temporary graphical model as follows. For each index node  $I$  in the Steiner tree, we compute the set of query variables that have been assigned to it, denote by  $I(V)$ . We also compute the quantity  $neighbors(I)$ , which represents the set of random variables that belong to the separators adjacent to node  $I$  in the Steiner tree. If  $I(V) = \phi$ , we determine if there is a shortcut potential  $P$  which contains all the variables present in  $neighbors(I)$ . In that case, we just marginalize the shortcut potential to include only  $neighbors(I)$  and return it. Otherwise, we recurse along that node with query variables  $I(V) \cup neighbors(I)$ . After constructing the temporary graphical model, we eliminate the non-query variables from it and return the joint distribution over the query variables.

The algorithm for extraction queries is almost identical to that of inference queries, the only difference being that we do not execute step 18 of the algorithm described above, i.e., we do not eliminate the non-query variables inside

the recursion.

---

**Algorithm 1** query(inode,vars)

---

```

1: for i = 1 to vars.length() do
2:   found[i] = search(vars[i], inode.children)
3: if  $\forall i$ , found[i] = c then
4:   if inode.children[c].type = separator then
5:     return p(vars) from the separator clique
6:   else
7:     return query(inode.children[c], vars)
8: else
9:   Tree t = SteinerTree(inode.childTree, found)
10:  Initialize: GraphicalModel gm = null
11:  for every index node I in t do
12:    nrs = neighbors(I)
13:    I(V) = query variables in I
14:    if  $I(V) = \phi$  &  $\exists$  shortcut  $P$  s.t.  $nrs \in P$  then
15:      gm.add(I.shortcutpotential(nrs))
16:    else
17:      gm.add(query(I, I(V)  $\cup$  nrs))
18:  Eliminate non-query variables from gm & compute probability distribution p(vars)
19:  return p(vars)

```

---

**Example:** Suppose we are given the inference query  $\{e,o\}$  on the junction tree in Figure 2.2(c). We now describe the sequence of steps followed in the recursive procedure. In the first step, we discover that  $e \in I_1$  and  $o \in I_3$ , hence we determine the Steiner tree joining  $I_1$  and  $I_3$ . This is shown in Figure 5.3(a). After this, we pose the query  $\{e,c\}$  on node  $I_1$ , query  $\{c,j\}$  on node  $I_2$  and  $\{j,o\}$  on node  $I_3$  to continue the recursion. When the query  $\{e,c\}$  is posed on  $I_1$ , we again compute the Steiner tree joining the cliques containing  $e$  and  $c$ , shown in Figure 5.3(b), after which the query  $\{a,c\}$  is posed on partition  $P_1$  and  $\{a,e\}$  is posed on partition  $P_2$ . When the query  $\{c,j\}$  is posed on node  $I_2$ , we discover that it is present in the shortcut potential of the root and hence,

we can directly compute the probability distribution of  $\{c,j\}$ . When the query  $\{j,o\}$  is posed on the node  $I_3$ , we obtain the Steiner tree shown in Figure 5.3(c), following which we pose query  $\{j,l\}$  on  $P_5$  and query  $\{l,o\}$  on  $P_6$ . The final graphical model computed for the corresponding extraction query is shown in Figure 5.3(d). Notice that the graphical model is much smaller than the one shown in Figure 2.5 (which was constructed without the index).

### 5.3.2 Aggregate Queries

Aggregate queries are specified using a set of variables  $S$  and the aggregate function  $f$ . Our aggregate semantics is based on possible world semantics. Suppose that  $f$  is MIN. In each possible world, values are assigned to all the random variables, and we determine the value of the minimum in each world. Then we sum up the probabilities of all the worlds which yield this value to the minimum and compute the probability distribution of the minimum. We currently support *decomposable aggregates* -  $f$  is a decomposable aggregate for a set of random variables  $S = \{s_1, s_2, \dots, s_n\}$ , if it satisfies the following condition.

$$f(s_1, s_2, s_3, \dots, s_n) = f(f(s_1, s_2), s_3, \dots, s_n)$$

Informally, if we can apply the aggregate function piece by piece incrementally over the set of random variables, then the aggregate function is decomposable. In previous work [61], we showed how to exploit decomposability of aggregates to efficiently execute aggregation queries for the special case of Markovian sequences. Here, we develop an extension of that technique for probabilistic data with arbitrary correlations.

The naive method of executing aggregate queries is by first running the ex-

traction query, thereby obtaining the graphical model containing all the input variables and in the second step, constructing the graphical model corresponding to the aggregate function and inferring the value of the aggregate. However, this approach does not exploit the conditional independences that might exist among the input variables. Instead we propose the following approach, where we *push* the aggregate computation inside the index.

We describe the intuition behind our algorithm for aggregation by illustrating it with an example. Suppose we want to compute the sum of the values of the attribute  $V_1$  in the relation  $R_1$  in Figure 2.2(a). This corresponds to computing the aggregate of the random variables  $\{b,c,d,k,n,o\}$ . In the naive method, we run an extraction query over these random variables and extract a junction tree containing these variables. However, the junction tree extracted in this case is almost as big as the original junction tree. On carefully analyzing the graph, we see that  $b, c,$  and  $d$  are independent of  $k, n,$  and  $o$  given the value of  $c$ . Similarly  $n$  &  $o$  are independent of  $k$  given the value of  $j$ . Suppose we first define random variables  $Y_1 = b + c + d$ ,  $Y_2 = k$  and  $Y_3 = n + o$ . Then, if we know the distributions of each of these random variables along with the separators, i.e.,  $p(Y_1, c)$ ,  $p(c, Y_2, j)$  and  $p(j, Y_3)$ , then we can construct the aggregate value exactly from these functions. In essence, our algorithm is going to “push” the aggregate computation inside the index, extracting only probability functions such as above.

The algorithm we have designed is a recursive algorithm just as for inference queries. We illustrate the working of our algorithm for the above query. In the first step, the algorithm determines that  $b, c$  and  $d$  are present in node  $I_1$  and that  $k$  is present in  $I_2$  and that  $n, o$  are present in  $I_3$ . A recursive call is made on  $I_1$  with two sets of parameters:  $\{b,c,d\}$  and  $\{c\}$  which means

that we need to compute and return the distribution between  $b + c + d$  and  $c$ , we denote this by  $\text{agg-inf}(I_1, \{b, c, d\}, \{c\})$ . Now, this induces further recursive calls on the partitions  $\text{agg-inf}(P_1, \{b, c\}, \{c, a\})$  and  $\text{agg-inf}(P_2, \{d\}, \{a\})$ . The final call is just an inference query on  $P_2$ . We perform the aggregation algorithm on the partition simply by first doing the inference query and then using the joint probability distribution function to determine the distribution of the aggregate. The results from  $P_1$  and  $P_2$  are then multiplied to obtain the probability distribution  $p(b + c, d, c)$ , which is then processed to obtain  $p(b + c + d, c) = p(Y_1, c)$ . The recursive call from  $I_2$  leads to an inference query  $\text{agg-inf}(P_3, \{k\}, \{c, j\})$ . Similarly the recursive call on  $I_3$  leads to two inference queries  $\text{agg-inf}(P_5, \{n\}, \{j, l\})$  and  $\text{agg-inf}(P_6, \{o\}, \{l\})$ , which are then processed as before to obtain the probability distribution function  $p(j, Y_3)$ . The final top-level junction tree that we obtain as result is shown in Figure 5.3(e). The size of this output graphical model constructed is much smaller than the naive model generated from the inference query, resulting in significant savings in query processing times.

## 5.4 Handling Updates

In this section, we describe the algorithms that we have developed for modifying the index in response to updates to the underlying probabilistic database. Our system supports the following two kinds of updates to the probabilistic database.

- The first is a modification of the existing data, i.e., modification of a probability distribution, or the assignment of a deterministic value to an existing random variable. For instance, if we verify the occurrence of the tuple with



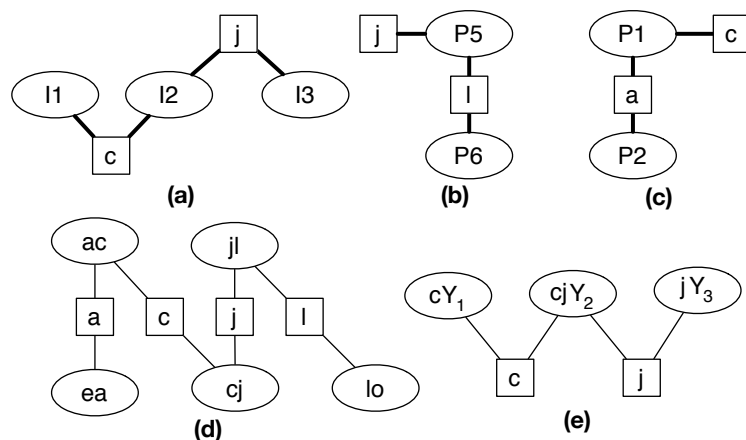


Figure 5.3: The Steiner trees generated at different index nodes while executing the inference query  $\{e,o\}$  on the junction tree in Figure 2.2(c) is shown in (a), (b), (c). (d) shows the final graphical model generated as a result of the extraction query  $\{e,o\}$ . The junction tree generated by the aggregation query is shown in (e)

$gid=2$  in the probabilistic database of Figure 2.2(a), then we need to set the value of random variable  $a$  to 1 in the database.

- The second is an insertion/deletion of a new tuple into the probabilistic database. This occurs when we need to add a new compound event, which is correlated with already existing events in the database. Here, we need to construct a new clique for the new random variable and add it to the database.

### 5.4.1 Updates to Existing Potentials

Updating the potential of a random variable  $v$  requires us to appropriately modify the potentials of all the cliques in the junction tree. The naive technique for updating a junction tree involves the message passing algorithm in which we transmit the knowledge of the update to every node in the tree through messages that are sent from the modified node to every other node. In

the first step, we identify a clique, say  $C$ , to which the random variable belongs and modify its clique potential to reflect the knowledge of the update. In the next step, the clique sends out a message to inform all of its yet uninformed neighbors about the update. Each of the neighbors then uses the message received, updates its potential and recursively sends messages to its neighbors; the process continues until all of the nodes have the knowledge of the update. After having completely updated the cliques in the junction tree, we can now update the shortcut potentials of every index node in the database. Since this algorithm spans the entire junction tree, it is clearly infeasible to perform this for large trees for every new update. Instead, we exploit the presence of shortcut potentials to develop a lazy strategy for efficiently updating both the index and the junction tree. This enables a *pay-as-you-go* framework in which future queries over the probabilistic data bear the cost for the updates. We illustrate our approach below.

In the first step, we use the index structure to efficiently identify a clique that contains the random variable to be updated and the partition containing it. Suppose we receive an update for a variable in the partition  $P$ . We load that partition into memory and perform the message passing algorithm over  $P$  alone and determine the correct probability distributions for every clique in  $P$ . In addition, we also update the shortcut potential of  $P$  based on the HUGIN algorithm (Chapter 2). Next, we load the parent node  $I$  of  $P$  and update the shortcut potentials of all the children of the node  $I$  and the separator potentials stored in  $I$ . We then load  $I$ 's parent and continue the same process recursively until we reach the root node. Updating the rest of the index and the junction tree is carried out whenever we get new queries on the database. When a query is posed on an index node, it verifies that the separator potentials and

the shortcut potentials stored in the index node is up-to-date. Otherwise, it updates them first using the message passing algorithm and then continues with the query processing. We note here that each query only updates those index nodes and only those partitions that are required for computing the answer to the query. We illustrate our algorithm with the following example.

**Example:** Suppose we receive an update  $i = 0$  in our running example. We will now indicate the sequence of updates we perform for this case. In the first step, we locate and load partition  $P_4$  into memory, following which, we update the probability distributions of the cliques  $hi$  and  $fh$ . In the next step, we update the shortcut potential  $p(f)$  of partition  $P_4$ . We then load the index node  $I_2$  and using  $p(f)$ , we update the shortcut potential of  $P_3$ ,  $p(c, j, f)$  and the separator potentials  $p(g)$ ,  $p(j)$  and  $p(c)$ . After this, we determine the new shortcut potential of  $I_2$ ,  $p(c, j)$ . We then load the root node and determine the new shortcut potentials of  $I_1$  and  $I_3$ . Suppose we now receive a query on variable  $e$ . When we recurse along the index node  $I_1$ , we first update the shortcut potential of  $P_1$ ,  $p(a, c)$  and that of  $P_2$ ,  $p(a)$  and then load the partition  $P_2$  into memory. We then update  $P_2$  completely and determine the probability  $p(e)$  as required by the query. Note here that we have only updated the partition  $P_2$ . We did not even need to update the partition  $P_1$ , we just updated its shortcut potential. This provides us with an efficient approach for updating the index. The gains are even more substantial when the partitions are much larger.

## 5.4.2 Inserting New Data

We now consider the problem of adding new data tuples to the database. In our setting, this problem corresponds to the problem of adding new random variables to the junction tree, given its correlated variables. Formally, we are given a node  $X$  and the set of edges that connect this node to its correlated variables  $S = \{s_1, s_2, \dots, s_k\}$ , and a joint distribution of all the nodes in  $S \cup \{X\}$ . In the underlying PGM, this corresponds to just modifying the graph by adding a new node to the graph along with the edges. On the junction tree, we have to create a new clique node for the new variable  $X$  and update the cliques that are modified as a result of the addition of new edges.

We propose a two step process for this. In the first step, we modify the junction tree to reflect the addition of this new data tuple and in the subsequent step, we make the junction tree consistent using message passing using the lazy approach described in the previous section.

**Creating a new clique for the new node:** The algorithm first searches for the neighbor  $s_1$  of the new random variable using the index data structure. After loading the relevant partition into memory, it computes the relevant clique containing  $s_1$ . We make a new clique containing the new random variable and  $s_1$ . But we first need to assign an id for the new random variable introduced.

**Assigning a new Id to the new variable:** To add a new variable to the junction tree, we need to first issue a unique id to the variable. We can extend the range of the partition by 1 and assign this value to the new variable. But this does not work since another variable could already possibly have this id assigned to it. The alternative is to assign the id equal to one higher than the previously assigned highest variable id. However, assigning such a new id to

this variable results in the violation of the contiguous variable name property (See Section 5.1), i.e., the id of the new variable will exceed the max value of the range lists for this partition. In order to deal with this problem, while we assign ids to the variables after the hierarchical partitioning, we add *gaps* in the ranges between one partition to the next, these gaps act as holes for subsequent addition of newer variables. Also we increase amount of gap exponentially (in the number of children in the index structure) as we go to higher levels in the index structure (if we cross an index node). Whenever the gap between two partitions  $P_1$  and  $P_2$  is filled completely, we go to their parent index node and request more gap between the partitions and renumber the variables in the partition  $P_2$  to account for the newly inserted gap. If no more gaps are available in the parent, then we recursively go up the tree looking for a node that has sufficient gaps. Note that we also have to update the range lists and addLists for every index node that had its ids modified, hence we recursively update the index (range lists, add lists, separators) starting from the partition in which the new variable was inserted. Now, we modify the junction tree to reflect the addition of edges between the new variable and its neighbors.

**Adding neighbors:** To reflect the addition of the new neighbors of the variable, we use the following approach. For each neighbor  $s_i$ , we compute the shortest path joining the clique containing  $s_i$  to the new clique. We add the new variable to every clique and every separator along this path. In addition, we remove any clique that becomes a subset of a newly created clique. The resulting graph is a valid junction tree as shown in Berry et al. [11]. To update the index, we only need to add the new variable id to every partition along the path.

**Writing partitions back to disk:** As more and more insertions happen to our database, the sizes of the partitions will increase since the sizes of every clique that had a new variable inserted increases. Hence, whenever we write back an index node or a partition back to disk, we determine its size and if it exceeds the block size, we use the partitioning algorithm and split it into smaller subtrees which fit into a disk block. We construct a new index node in place of the disk block and accordingly assign the parent and child pointers.

### 5.4.3 Deletions

Deletions can also be viewed as insertions of new data elements. For instance, deleting a tuple  $X$  from the database is equivalent to adding a new boolean random variable  $V_X$  that specifies whether to consider  $X$  or not. For this, we connect the random variable  $V_X$  to every random variable which is connected to  $X$  in the PGM. We set  $V_X$  to zero to delete  $X$ , in case we need to insert  $X$  again, we set the variable back to 1. This particular method of deletion is not efficient since over time, deletions would continuously increase the size of the database. We are currently developing more efficient methods for deleting variables from the database.

## 5.5 Experimental Evaluation

### 5.5.1 Implementation Details

We implemented the INDSEP component of PrDB (Figure 3.1) to test its benefits on the query processing performance. As described in Chapter 3, INDSEP is implemented in main memory. The disk is simulated as an array

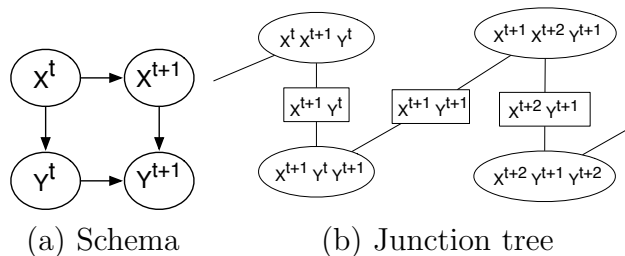


Figure 5.4: We generate the Markovian sequence database using the schema shown in (a). The junction tree structure of the Markovian sequence is shown in (b)

of disk blocks, each of size `BLOCK_SIZE`, a user defined parameter. `INDSEP` is stored in this virtual disk. Each index node (leaf or interior) is stored in a single disk block. We also implemented a *DiskManager*, a singleton module that runs constantly and manages the disk blocks. All accesses (read,write,modify) to the virtual disk is made via the `DiskManager`. This also enables us to keep track of the exact number of accesses made to the disk. We will use these counts heavily in demonstrating the benefits of `INDSEP` in the following section.

## 5.5.2 Experimental Setup

All of our experiments were carried out on a machine with a 2.4 GHz Intel Core 2 Duo processor and 2GB memory. We evaluate the performance of our index on the following two probabilistic databases.

- **General probabilistic database:** We generate a probabilistic database on 2 relations that is representative of a typical event monitoring application (see Section 1.1.1). The database contains a total of about 500,000 tuples corresponding to detected events. It exhibits attribute uncertainty, tuple uncertainty and tuple correlations. We simulate arbitrary correlations in

the PGM, by connecting each random variable to  $k$  neighbors, where  $k$  itself is randomly chosen between  $[1, 5]$ . We then construct the junction tree equivalent of the PGM and then bulk-loaded the database and the index blocks. We also allow continuous updates to the database corresponding to the new events being detected.

- **Markovian Sequence database:** We generate a Markovian sequence database [61] with schema shown in Figure 5.4(a). We bulk load the database with a total of 1 million time slices, which corresponds to 3 million nodes in the junction tree. Updates continue to occur periodically in the database with the new node  $X_{t+1}$  being inserted with neighbor  $X_t$ , and  $Y_{t+1}$  being inserted with neighbors  $Y_t$  and  $X_{t+1}$ .

#### Query & Update Workloads:

We generated 4 different workloads of queries based on the size of the spanning tree that needs to be constructed for executing the query. Each query is over 2 to 5 variables. Each workload consists of a total of 25 queries. We use the information from the partitioning of the tree in order to generate the workload.

- $W_1$ : Shortest-range queries. These are queries that have a span of about 20% of the junction tree.
- $W_2$ : Short-range queries. These have a span of 40% of the junction tree.
- $W_3$ : Long-range queries. These have a span of 60% of the junction tree.
- $W_4$ : Longest-range queries. Each query in  $W_4$  spans at least 80% of the tree.

We use each of the above workloads for both *inference* and *aggregate* queries. Similar to the above query workloads, we generate 4 different update workloads (for the first probabilistic database). Each newly added variable has a set of



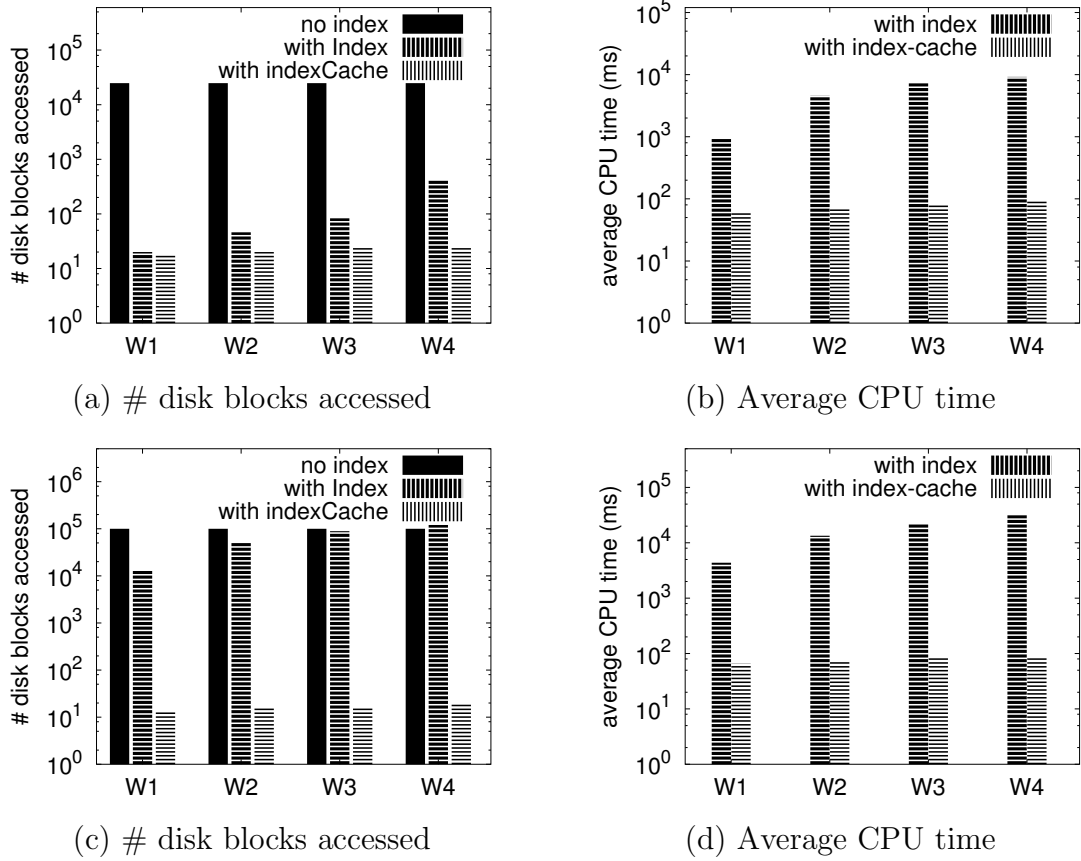


Figure 5.5: Illustrating query performance in terms of number of blocks accessed and cpu time for workloads  $W_1$ ,  $W_2$ ,  $W_3$  and  $W_4$  when index data structure is absent, index is present without shortcut potential, both index and shortcut potential are present. (a) & (b) correspond to the event database, while (c) & (d) correspond to the Markovian sequence database. We note that the graph is in *logarithmic scale*, so the gains are substantially more than what is apparent.

neighbors, the size of which is uniformly chosen between 2 and 4. Based on the distances between the neighbors, they are classified into 4 update workloads  $W_1$ ,  $W_2$ ,  $W_3$  and  $W_4$  just as described above.

### Comparison Systems:

We compare our *INDSEP* data structure against two other approaches:

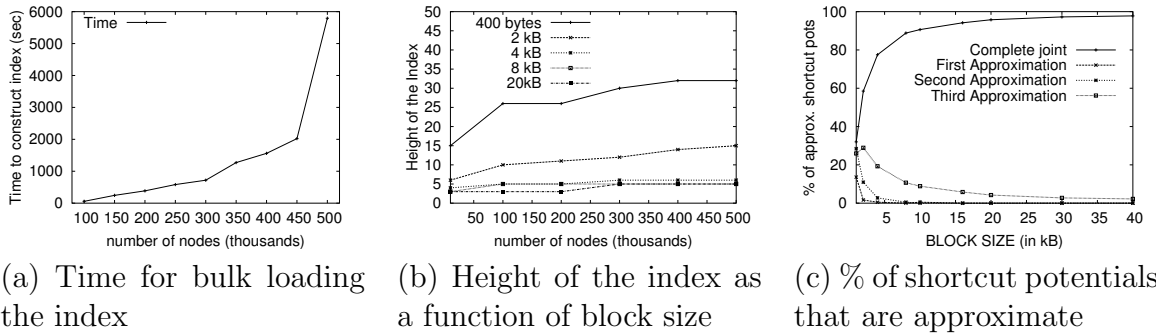


Figure 5.6: As shown in part (a), the time taken to bulk load the index is linear in the size of the database. Part (b) shows that the height of the tree increases in a logarithmic-like fashion as the size of the database increases. Part (c) shows that as the disk block size increases, the amount of approximation reduces, i.e., less than 20% for 4kB block size.

- **No index:** In this case, we do not maintain any indexes in the database and perform query processing using the naive technique described in Chapter 2.
- **Index without caches:** In this case we maintain the index over the junction tree, but do not maintain any shortcut potentials. The key advantage of this approach over the naive approach is that we can reduce the number of disk blocks accessed significantly, but the overall performance remains linear.

### 5.5.3 Results

#### Effectiveness of the Index:

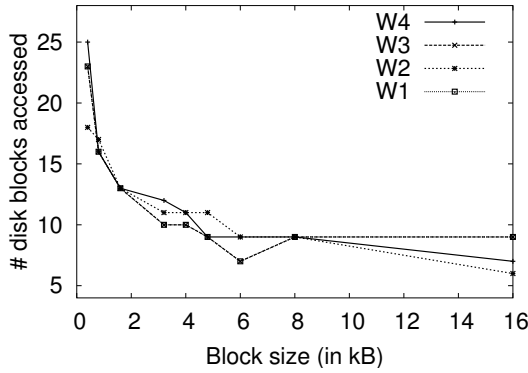
For our first experiment, we ran each of the query workloads  $W_1$ ,  $W_2$ ,  $W_3$  and  $W_4$  for the three comparison systems and computed the average number of disk blocks accessed in order to answer the inference query. We also measured the average wall clock CPU times for each of the workloads. We plot our results

as a bar graph in Figure 5.5(a) & (b). As shown in the figure, we obtain an order of magnitude improvement both in the number of disk blocks accessed as well as in the CPU cost. Notice that the y-axis is in logarithmic scale, so the gains are substantially more than what is apparent.

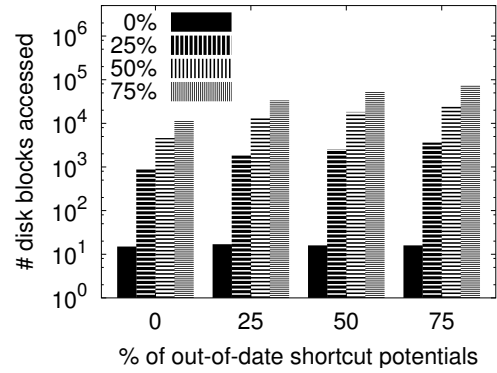
We also note the benefit of our *shortcut* potentials for workloads  $W_3$  and  $W_4$ , which are primarily responsible for reducing the number of disk blocks accessed and the CPU cost in this case. Using indexes alone does prove useful for short range queries in the workloads  $W_1$  and  $W_2$ , but for longer range queries, using shortcut potentials reduces the computational time even further. In fact, for the Markov Sequence database which generates a junction tree with very large diameter (graph-theoretic) of about a million, using just the index can actually be more expensive for long range queries as shown in Figure 5.5(c). The overhead occurs since the query processor needs to traverse every disk block in the database along with almost all the index blocks. Augmenting the index with shortcut potentials reduces the number of disk blocks accessed and the CPU time by more than a factor of 1000 (Figure 5.5(d)).

### **Study of the Index Structure:**

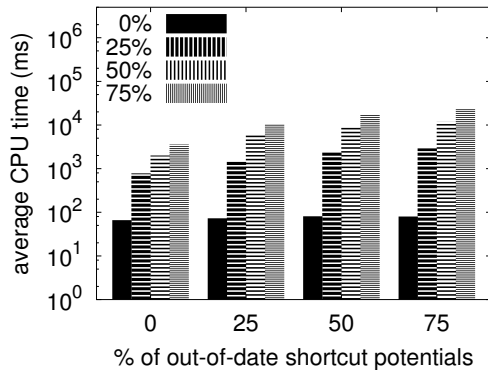
Here, we study the structure of the *INDSEP* data structure and provide details of its shortcut potentials. We first generate 10 different event datasets ranging from 50,000 cliques to 500,000 cliques and construct the index data structure for each of the data sets. We first measure the time take to construct the index as a function of the size of the database. Using a block size of 1 kB, we measure the amount of time it takes to fully construct the index. We plot our results in Figure 5.6(a). As shown in the figure, the time taken increases linearly as the size of the database increases as expected. We attribute the sudden jump



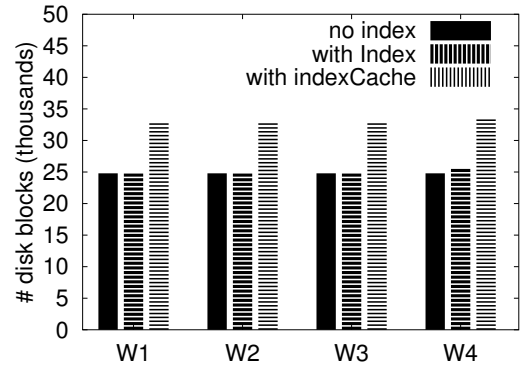
(a) Query performance as a function of block size



(b) Query performance as a function of % absent shortcut potentials (disk blocks)



(c) Query performance as a function of % absent shortcut potentials (cpu cost)



(d) Update performance

Figure 5.7: Graph in part (a) shows that the query performance (as function of number of blocks) improves when block size is increased. The bar graphs in parts (b) & (c) show that the query performance falls as the percentage of shortcut potentials that are out-of-date increases. Part (d) shows the update times – as we can see, the overheads for updating INDSEP data structure are quite minimal.

in the time taken to build the index for 500,000 nodes to thrashing.

We now determine the height of the hierarchical index structure as a function of both the size of the database and the block size. We use a range of block sizes starting from 0.4kB to 40kB. We plot the results in Figure 5.6(b). As shown in the figure, the height of the index structure increases with the size of the graph, but quite slowly. Also, for a reasonable block size of 4kB-8kB,

the height of the tree is about 7 even for quite a large tree of around 500,000 nodes.

We now study the structure of the shortcut potential, i.e., we want to identify the percentage of shortcut potentials in the index structure that use approximations, as a function of block size. We construct our index data structure on a junction tree of size 500,000 nodes for different values of block size and compute the percentage of index nodes that store the complete joint distribution (no approximation), the first, second and third approximations to the shortcut potentials. The results are plotted in Figure 5.6(c). As shown in the figure, for smaller values of block sizes below 4kB, only about 40% of the index nodes contain the full joint distribution, while larger block sizes allow many more index nodes to store the complete joint distribution in their shortcut potentials. For a reasonable block size between 4-8kB, less than 20% of the index nodes approximate their shortcut potentials.

### **Study of Query Processing Performance:**

In this section, we take a closer look at the performance of inference queries for different values of the index parameters. We first vary the block size and analyze the performance of the query for each case. We used block size values between 0.4 kB and 16 kB. The results are plotted in Figure 5.7(a). As the size of each disk block increases, we observe that the number of disk blocks that needs to be accessed reduces as expected, but it remains fairly constant after the block size exceeds a certain size.

In the next experiment, we study the effect of the shortcut potentials on the query performance. As described earlier, updates to a variable need to be propagated to the entire database. In our lazy update implementation, we

modify only the shortcut potentials of certain nodes in the tree while updating the other potentials on demand from the queries. To formally study this case, we arbitrarily set  $x\%$  of the shortcut potentials in the Markovian sequence database to be out of date and then measure the query processing performance as a function of  $x$ . We plot the results in Figure 5.7(b) & (c). As shown in the Figure, as the value of  $x$  increases, more blocks need to be updated by the query which results in drop in the query performance. But we note here that once the first query subsequent to the update, updates the index nodes and shortcut potentials relevant to it, further queries that access the same data can again use the valid shortcut potentials and obtain a performance closer to the ones shown in Figure 5.5(a),(b).

### **Study of Update Performance:**

Here, we study the performance of index when the database is subjected to updates. For each of the update workloads, we determine the average number of disk blocks that need to be read and modified in order to completely update the database (not the lazy version). We compare the performance of our *INDSEP* data structure with the comparison systems (1) & (2). We plot the results in Figure 5.7(d). As shown in the figure, with minimal overhead, we can update the index data structure - particularly the add lists, this is indicated by the middle bars. Updating the shortcut potentials requires us to read in all the index blocks, since all the shortcut potentials need to be updated, which is also quite small compared to the size of the database.

# Chapter 6

## Lineage Processing over INDSEP

In this chapter, we develop algorithms for evaluating conjunctive queries over correlated probabilistic databases. As we showed in Chapter 2, we follow a 2 step process: in the first step, we determine the lineage of the output tuples, which is a boolean formula that denotes the condition for the existence of the output tuple, and in the second step, we compute the probability of the lineage formula. The outline of the chapter is as follows. In the first part of the chapter (Section 6.1), we develop algorithms for computing the probability of lineage formulas over junction trees. Following this (Sections 6.2,6.3), we describe how to scale these algorithms to large-scale probabilistic databases via the INDSEP data structure which we developed in Chapter 5. We conclude the chapter with an experimental evaluation of the proposed algorithms in Section 6.4.

## 6.1 Lineage Processing Algorithms over Junction trees

In this section, we develop algorithms for processing lineage formulas over junction trees. Although our focus is on read-once lineages, our algorithms for lineage processing can be applied even to non-tree structured lineages. While it has already been shown that read-once lineages can be processed in polynomial time for tuple independent probabilistic databases [27, 96], we show that the problem of processing them on *lightly correlated* probabilistic databases is #P-complete.

**Theorem 1.** *Processing read-once lineages on correlated junction trees is #P-complete. In fact, processing them on Markov chains is also #P-complete.*

*Proof.* (Sketch) We start with the problem of counting the number of satisfying assignments to a monotone DNF formula  $\phi$ , which is known to be #P complete [107]. Convert  $\phi$  into a read-once formula, by replacing each repeating literal in  $\phi$ , by a new literal. Suppose the new formula obtained is  $\phi'$ . Keep track of the equivalent literals. Now, construct a graphical model for  $\phi$ , - create a node for each literal in  $\phi'$  and create a node for each of the clauses. Finally, create a node for the  $\phi'$  value itself. For each literal, draw a directed edge from the node containing the literal to the clause that contains the literal (Since  $\phi'$  is read-once, each literal is contained in exactly once clause). In addition, draw 1 edge between each literal and its successive alias (Note that we are creating a chain among all the “equal” literals and not a clique). Assign factors to the nodes of the graphical model as follows. To each leaf, assign the factor [1 0.5, 0 0.5]. To each of the OR clause nodes, assign an OR



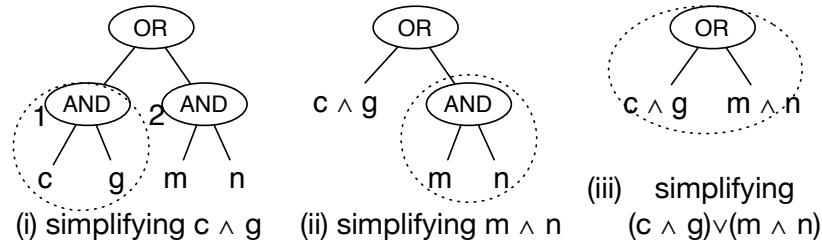


Figure 6.1: Different stages in the simplification process

factor and to each of the AND clauses, assign an AND factor (use associative property to decompose the clauses if required). Finally to each of the equality nodes, assign equality factors  $[1 \ 1 \ 1, 0 \ 0 \ 1]$ . It is quite easy to see that the above construction is polynomial in the size of the input. Now, we compute the marginal probability of the  $\phi'$  node. The solution to the original problem can be obtained by multiplying  $p(\phi')$  by  $2^N$ .  $\square$

However, most real world datasets do not exhibit this worst case behavior. The junction trees are often disconnected, which allows us to perform exact computation.

### 6.1.1 Message Passing for Lineage Processing

Before discussing the message passing algorithms, we introduce a parameter called *width* that captures the complexity of processing a lineage formula. We illustrate *width* using the naive method we described in Section 2.1.5. Suppose that we want to compute the probability of the lineage formula  $(c \wedge g) \vee (m \wedge n)$ . In the first step, we evaluate the inference query  $\{c, g, m, n\}$  and determine  $p(c, g, m, n)$ . In the next step, called *simplification*, we evaluate the probability of the lineage. As described earlier, we first multiply the pdf with  $p(c \wedge g | c, g)$  to get  $p(c, g, m, n, c \wedge g)$  and then eliminate  $c$  and  $g$  to get

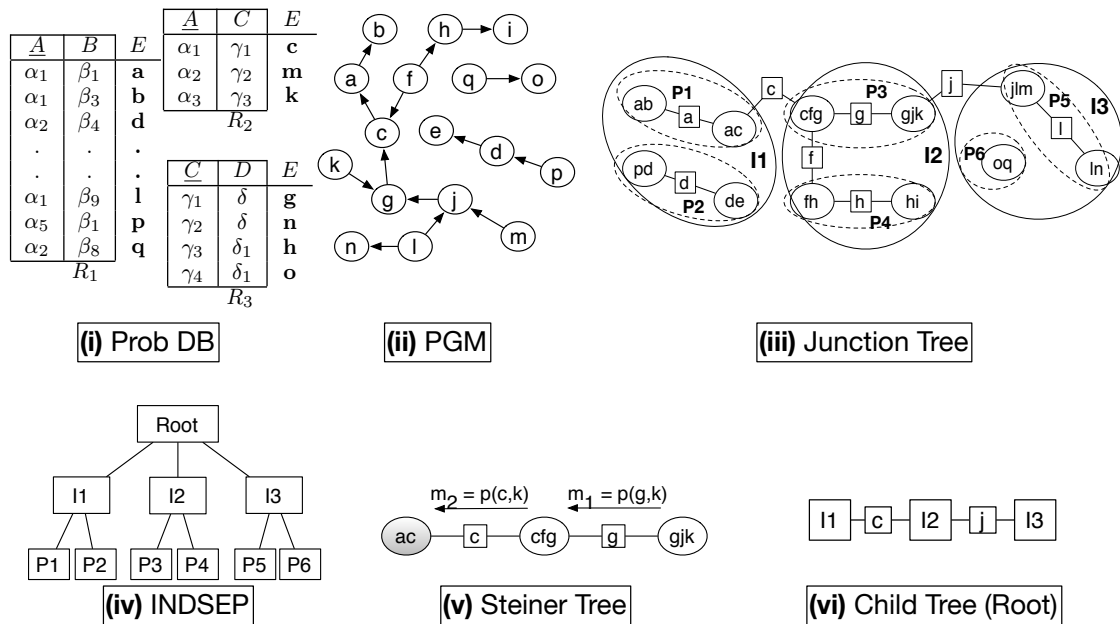


Figure 6.2: Figure shows (i) a tuple uncertain probabilistic database (ii) the graphical model that captures the correlations among the various tuples (iii) its equivalent junction tree (iv) the INDSEP data structure corresponding to the junction tree in (iii) and (v) Steiner tree computed while evaluating the inference query  $\{a, k\}$ . Pivot clique  $(ac)$  is shaded. (vi) The childTree stored in the root is shown here. Note that  $I_2$  is connected to  $I_1$  via  $c$  and to  $I_3$  via  $j$  as indicated in (iii). Note that (i),(ii) and (iii) are repeated from Chapter 2 for convenience.

$p(c \wedge g, m, n)$ . We proceed in similar fashion to compute  $p(c \wedge g, m \wedge n)$ , and finally the probability  $p((c \wedge g) \vee (m \wedge n))$ . We pictorially illustrate the simplification process in Figure 6.1. The above algorithm creates several intermediate pdfs, the size of the largest intermediate pdf being  $2^5 = 32$ . The time taken by the algorithm is influenced by the size of the maximum pdf created. We call this parameter as the *width* induced by the lineage formula. Although *width* has superficial similarities with the graph theoretic *treewidth* [94], we distinguish them since *width* depends on the actual algorithm we use, unlike *treewidth*. Also, *treewidth* corresponds to the optimal junction tree and is

in general NP-hard to compute exactly [94].  $Lwidth$  can be considered as an upper bound to treewidth. The size of the biggest probability distribution is  $2^{Lwidth}$  and hence, the computational time is exponential in the  $Lwidth$  induced by the formula. Next, we develop an improved recursive algorithm based on message passing. The trick is to create smaller intermediate pdfs by performing the simplifications eagerly, whenever we detect a message that can be simplified. We illustrate our Eager strategy by showing how to reduce the  $Lwidth$  of the above formula from 5 to 4.

**Improvement 1 (Eager):** As with the HUGIN algorithm (Chapter 2), we first select a pivot and construct a Steiner tree over the variables referred in the lineage formula. At each clique, we multiply all the incoming messages and the clique's pdf to get an intermediate pdf, and then eliminate the non-query variables. However, we try to simplify the pdf as much as possible based on the given input lineage, before sending the resulting pdf as a message. Now, we compute  $p((c \wedge g) \vee (m \wedge n))$  using the Eager strategy. We will use the same probabilistic database from Chapter 2, it is repeated in Figure 6.2(i) for convenience. Suppose that we select as pivot, the clique (cfg). The Steiner tree for this lineage is the path connecting the clique (ln) to the clique (cfg) in Figure 6.2(iii). The algorithm proceeds as follows:

- The clique (ln) sends message  $p(l, n)$  to the clique (jlm).
- Now, clique (jlm) multiplies the incoming message  $p(l, n)$  with its pdf  $p(j, l, m)$  to get  $p(j, l, m, n)$  (it also divides by  $p(l)$ ). It eliminates  $l$  since it is no longer required to get  $p(j, m, n)$ . Since  $m$  and  $n$  are together (and do not appear elsewhere in the junction tree), it simplifies  $p(j, m, n)$  here itself, to get  $p(j, m \wedge n)$  and sends it to clique (gjk).

- Clique (gjk) multiplies  $p(j, m \wedge n)$  with its pdf  $p(g, j, k)$ , eliminates  $k$  &  $j$ , sends  $p(g, m \wedge n)$  to clique (cfg).
- Clique (cfg) eliminates  $f$  from its pdf  $p(c, f, g)$  to get  $p(c, g)$  and multiplies it with  $p(g, m \wedge n)$  to get  $p(c, g, m \wedge n)$ . After relevant simplifications, the clique (cfg) computes the final result  $p((c \wedge g) \vee (m \wedge n))$ .

In this approach, the maximum intermediate pdf size generated is  $2^4$ . This reduction is small for the above toy example, but it can be very large for larger lineages, since the computational complexity is exponential in the lwidth.

We can reduce the lwidth induced by the lineage even further by performing the simplification even before we multiply all of the incoming messages at a clique node. Suppose that we want to compute the probability of another boolean formula  $(c \wedge h) \vee (m \wedge n)$  and we pick the clique (cfg) as the pivot. We can use the Eager strategy as before and determine the messages and the intermediate probability distributions that will be generated during the algorithm. Consider the last step in the operation in which the message  $p(g, m \wedge n)$  is sent to (cfg) via the separator  $g$  and the message  $p(f, h)$  is sent to (cfg) via the separator  $h$ . In the Eager strategy, we first multiply  $p(c, f, g)$ ,  $p(f, h)$  and  $p(g, m \wedge n)$  to get  $p(c, f, g, h, m \wedge n)$ . Then we eliminate  $f$ ,  $g$  and simplify it to get  $p((c \wedge h) \vee (m \wedge n))$ . The biggest pdf created by this strategy is of size  $2^5$ .

Alternatively, we can multiply the three pdfs incrementally and simplify when possible, i.e., we first multiply the pdfs  $p(c, f, g)$  and  $p(f, h)$  to get  $p(c, f, g, h)$ . Then, we eliminate  $f$  and simplify to get  $p(g, c \wedge h)$  (Note that we can do this since both  $c$  and  $h$  do not appear in the remaining message  $p(g, m \wedge n)$ ). Now, we can multiply  $p(g, c \wedge h)$  with  $p(g, m \wedge n)$  to get

$p(c \wedge h, m \wedge n, g)$ , eliminate  $g$  and simplify to get the output. In this case, we would only create a pdf of size  $2^4$ . Note that the *ordering* of multiplications is important: if we had chosen to multiply  $p(c, f, g)$  and  $p(g, m \wedge n)$  first, we would not be able to reduce the lwidth. Next, we present a heuristic to select a good ordering.

**Improvement 2 (Eager+Order):** We construct a complete graph in which each node corresponds to the probability distribution which is to be multiplied. We then set the weights to each of the edges in the graph as follows. The weight of an edge is equal to the *amount of simplification* that is possible if we multiply the pdfs corresponding to its adjacent nodes. The amount of simplification while multiplying two probability distributions  $f_1$  and  $f_2$  is given by  $|f_1 \cup f_2| - |f|$ , where  $f$  is the simplified output after multiplying  $f_1$  and  $f_2$ . For instance, in the example above, when we multiply  $p(c, f, g)$  and  $p(f, h)$  the final output is  $p(g, c \wedge h)$ , hence the simplification is given by  $4 - 2 = 2$ . We greedily pick the edge with the largest weight and multiply the probability distributions together. We then perform simplification and update the graph, by clustering the 2 nodes together and recomputing the weights of all edges incident on the newly created node. We continue this process until all the probabilities have been multiplied, i.e., when there are no more edges in the graph. The order of multiplication for the above example is illustrated in Figure 6.3(a). The edges that are selected by the heuristic are darkened. In the first step, we multiply  $p(c, f, g)$  and  $p(f, h)$  to obtain  $p(g, c \wedge h)$ . In the second step, there is only one edge left, hence we multiply this with  $p(g, m \wedge n)$ .

### 6.1.2 Pivot Selection

Another factor influencing the lwidth induced by the lineage formula is the pivot selected by the algorithm. Suppose we want the probability of  $(b \wedge c) \vee g$ . The Steiner tree corresponding to this query is shown in Figure 6.3(b). As shown in the figure, there are 3 choices of pivot selection, i.e., one of (ab), (ac) or (cfg). We will evaluate the lwidth for two different pivot locations - clique (ab) and clique (cfg).

**Case 1: Pivot = (ab):** The sequence of messages passed in this case are indicated in Figure 6.3(b). Clique (cfg) sends message  $p(c, g)$  to clique (ac). Now, (ac) multiplies it with its pdf  $p(a, c)$  to get  $p(a, c, g)$ , which is sent to clique (ab). (ab) multiplies  $p(a, c, g)$  with  $p(a, b)$  to get  $p(a, b, c, g)$ . Then it eliminates  $a$  to get  $p(b, c, g)$  from which we get  $p((b \wedge c) \vee g)$ . The maximum intermediate pdf for this pivot location is  $2^4$ .

**Case 2: Pivot = (cfg):** In this case, the clique (ab) sends the message  $p(a, b)$  to clique (ac). Now, (ac) multiplies it with its pdf  $p(a, c)$  to obtain  $p(a, b, c)$ . It also eliminates  $a$  since it is not required and simplifies  $p(b, c)$  to  $p(b \wedge c, c)$ , which is then sent to clique (cfg). (cfg) first computes  $p(c, g)$  by eliminating  $f$  from its joint pdf and then multiplies with  $p(b \wedge c, c)$  to get  $p(b \wedge c, c, g)$  which is then simplified to the result  $p((b \wedge c) \vee g)$ . Note that in this case, the maximum pdf size generated in this case is just  $2^3$ .

Hence, given a lineage formula, we need to come up with the optimal pivot location in order to process it efficiently. Since there are only  $n$  choices, ( $n$  is the number of clique nodes in the Steiner tree) for the pivot position, we use the naive approach in which we measure the *lwidth* induced for each pivot location. We then select the node which induces the smallest lwidth as the

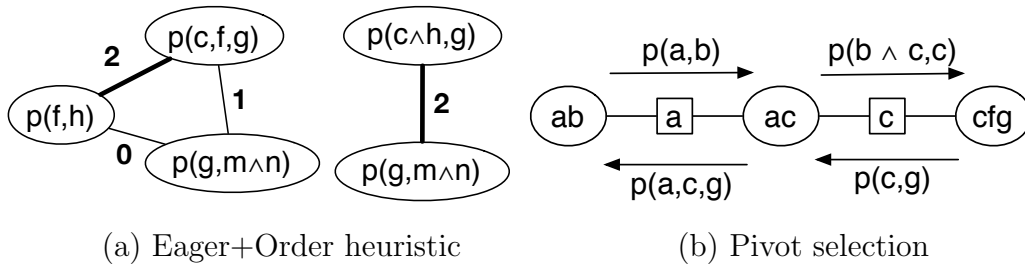


Figure 6.3: (a) Illustrating the order of multiplication and simplification in Eager+Order heuristic. Initially, we multiply pdfs  $p(f, h)$  and  $p(c, f, g)$  since that edge has the maximum weight. (b) When pivot = (cfg), the sequence of messages passed is indicated above the graph (right arrows). When pivot = (ab), the sequence of messages is indicated below the graph (left arrows).

pivot.

### 6.1.3 Dealing with Disconnections

Until now, we have assumed that the junction tree is a single tree that connects all the variables. However, the random variables may be correlated as a forest of junction trees. Here, we adapt our lineage processing algorithms to deal with these disconnections. In the first step, we split the query into subqueries over each of the components in the junction forest. For instance, consider the query  $Q = (d \vee e) \wedge (b \vee c \vee q)$ . We see that the variables in the lineage formula belong to three different connected partitions  $\{d, e\}$  in one partition  $P_2$ ,  $\{b, c\}$  in the second partition,  $P_1$  and  $q$  in the third partition  $P_6$ . Hence, we split  $Q$  into three subqueries, one for each connected partition. However, we see that instead of posing an inference query  $\{d, e\}$  on  $P_2$ , we can actually pose a lineage query  $d \vee e$  on  $P_2$ . Similarly, we can pose query  $b \vee c$  on  $P_1$  and query  $q$  on  $P_6$ . After executing each query independently on each of the components in the junction forest, we get 3 pdfs, namely  $p(d \vee e)$ ,  $p(b \vee c)$  and  $p(q)$ . We combine the result pdfs together using the Eager+Order

heuristic as described before.

### Comparison with Special Purpose Techniques

We now discuss the special case in which every variable in the junction tree is independent of the other variables, i.e., our database is a tuple independent probabilistic database. Since there are a number of probabilistic databases that perform lineage processing over tuple independent probabilistic databases, we now perform a comparison of the query processing techniques in the two approaches. Given a *read-once* lineage formula of size  $k$ , since our heuristic Eager+Order constructs a complete graph on  $k$  nodes and then uses the edge weights to decide the order of multiplication, the complexity of the operation is  $O(k^2)$ . However, existing special purpose techniques such as Mystiq [27] & Sen et al. [96] can perform lineage processing in  $O(k)$  time. This factor of  $k$  corresponds to the overhead involved in supporting correlated probabilistic databases.

## 6.2 Lineage Processing using INDSEP

In the previous section, we described efficient techniques for processing lineage formulas on junction trees. However, they do not scale to large junction trees, since performing a lineage query over few variables may require the algorithm to access the entire junction tree [62]. Hence, we use the recursive query processing framework called INDSEP, developed by Kanagal et al. [62], to scale our lineage processing algorithms. As we show in the experimental study, using INDSEP is not only advantageous for performance, but also for improving accuracy of our approximations.



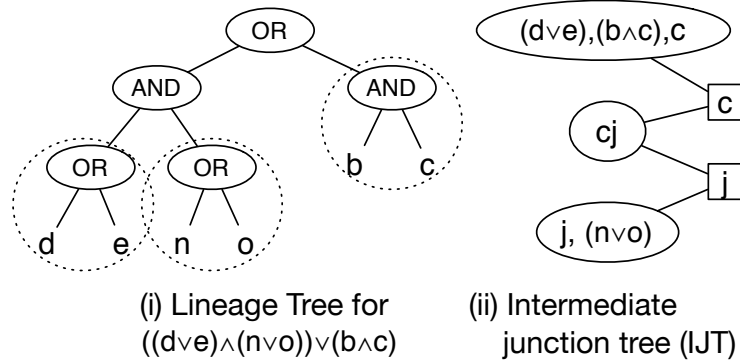


Figure 6.4: (i) illustrates the computation of subexpressions (ii) shows an intermediate junction tree generated in the root node while processing  $((d \vee e) \wedge (n \vee o)) \vee (b \wedge c)$

### 6.2.1 Recursive Approach

Recall that INDSEP is a hierarchical, tree-like data structure built on top of a junction tree (or a forest of junction trees). Lineage processing on INDSEP, analogous to any hierarchical index proceeds by recursion. Now, we describe the key recursion step for processing lineage formulas. During lineage processing, each index node involved is given as input, a set of expressions *ExpressionSet* which has two types of expressions contained in it: (1) Lineage formulas, which we denote by  $\vec{\lambda}$ , (2) Singleton random variables, which we denote by  $\vec{V}$ . The Index node is required to compute as output, the joint probability distribution between the expressions in the set, i.e.,  $p(\vec{\lambda} \cup \vec{V})$ . The complete lineage processing algorithm is shown in Algorithm 2. We explain the algorithm using a simple example.

Suppose we need to compute the probability of lineage formula  $\lambda = ((d \vee e) \wedge (n \vee o)) \vee (b \wedge c)$  (shown in Figure 6.4(i)) over the INDSEP data structure shown in Figure 6.2(iv). In the first step, we determine the random variables contained in  $\lambda$ , in our case this is  $\{b, c, d, e, n, o\}$ . For each variable here,

---

**Algorithm 2** process\_lineage(inode,  $\vec{\lambda}$ ,  $\vec{V}$ )

---

```
1:  $qvars = \vec{V} \cup variables(\vec{\lambda})$ 
2: for all  $v \in qvars$  do
3:    $found[v] = search(v, inode.vars)$ 
4: Graph  $tree = inode.childTree.Steiner\_tree(found)$ 
5:  $\{found = \text{set of child inodes that contain } qVars\}$ 
6: JunctionTree  $jtree = \text{null}$ 
7: for each  $node \in tree$  do
8:    $lvars = node.vars \cap vars(\vec{\lambda})$ 
9:    $ivars = node.vars \cap \vec{V}$ 
10:   $nrs = \text{neighboring separator variables of } node$ 
11:  if  $lvars = \phi$  then
12:    if  $ivars = \phi$  then
13:       $jtree.add(inode.shortcutPotential(nrs))$ 
14:    else
15:       $jtree.add(inference(nrs \cup ivars))$ 
16:  else
17:     $\vec{\lambda}' = \text{getSubExp}(\vec{\lambda}, lvars)$ 
18:     $\vec{V}' = nrs \cup ivars \cup (lvars - vars(\vec{\lambda}'))$ 
19:    if  $node$  is a leaf then
20:       $jtree.add(node.junctionTree.process\_lineage(\vec{\lambda}', \vec{V}'))$ 
21:    else
22:       $jtree.add(process\_lineage(node, \vec{\lambda}', \vec{V}'))$ 
23: return  $jtree.process\_lineage(\vec{\lambda}, \vec{V})$ 
```

---

we search for the child index node to which it belongs (pick arbitrarily if a variable belongs to multiple child nodes) (Steps 1-3). We then collect the variables present in each of the child nodes (Steps 8-9). In our example, the child node  $I_1$  contains the set of random variables  $\{b, c, d, e\}$  and  $I_2$  contains the set  $\{n, o\}$ . We now construct a Steiner tree over the childTree of the node (Section 5.1), joining all the child nodes which contain query variables (Step 4). In our example, we construct the Steiner tree connecting  $I_1$  and  $I_3$ , over the childTree of the root (Figure 6.2(vi)). Now, we need to determine the recursive calls to be made over the nodes in the Steiner tree, i.e., the ExpressionSet ( $\vec{\lambda}' \cup V'$ ) that needs to be posed to continue the recursion.

**Determining Recursive Calls:** We scan the set of random variables allotted to a child node and check if we can group the variables present in the child to form subformulas of the input lineage. These are added to  $\lambda'$  (getSubExp, Step 17). The remaining variables, which could not be grouped are collected in  $V'$ . In our example, in child  $I_1$ , the random variables  $b, c, d, e$  can be grouped into  $\{d \vee e, b \wedge c\}$  (Figure 6.4(i)) and is therefore added to  $\vec{\lambda}'$ , and there are no more variables, hence  $V'$  is empty. Note that we still need to capture the correlations among the variables  $b \wedge c, d \vee e$  and the rest of the variables in the query. Hence, we add the list of variables in the relevant separators of the child node to  $V'$  to capture all correlations (Step 15, 18). In our example, we add random variable  $c$  to  $V'$  since it is the separator of  $I_1$  (Figure 6.2(vi)). Hence, the complete ExpressionSet for child  $I_1$  is given by  $\{b \wedge c, d \vee e, c\}$ .

**Recursion:** Now, we proceed with recursive calls on the child nodes using the ExpressionSet assigned to them. In the special case when the ExpressionSet contains only separator variables, we can obtain the probability distribution directly from the shortcut potential (Step 13). In our example, since  $I_2$  does not contain any query variables, the only variables added to its ExpressionSet are its separator variables (Figure 6.2(vi)), given by  $\{c, j\}$ . This can be answered directly using the shortcut potential of  $I_2$ . To bottom out the recursion at the leaf nodes of the index, we use the algorithm of Section 6.1 to process the issued ExpressionSet over the junction trees contained in the leaf nodes (Step 20). Although the algorithms in Section 6.1 were designed for a single lineage formula, we can adapt them to process ExpressionSets easily, by ensuring that we do not eliminate the random variables that belong to other terms in the set.

**Assembling Child Results:** After obtaining the results from the child nodes, they are assembled as a junction tree - we call this as the *intermediate* junction tree (IJT). We now evaluate the remaining portion of the lineage over the IJT and return the result to the parent node. In our example, the child node  $I_1$  returns  $p(b \wedge c, d \vee e, c)$ ,  $I_2$  returns  $p(c, j)$ ,  $I_3$  returns  $p(j, n \vee o)$ , which is assembled as the junction tree shown in Figure 6.4(ii). Note that the cliques in the IJT contain newly created boolean variables  $d \vee e$ ,  $n \vee o$  and  $b \wedge c$ . We now compute the probability of  $((d \vee e) \wedge (n \vee o)) \vee (b \wedge c)$  over this junction tree using the algorithms of Section 6.1.

### 6.2.2 Shortcomings

Although the above algorithm works correctly, it has a few shortcomings which we describe here.

**Feasibility:** The complexity of the above algorithm is not entirely evident from the algorithm itself and is highly dependent on the nature of the underlying junction tree, and the structure of the lineage formula. If the random variables in the junction tree are independent, the algorithm runs quickly even for large lineage formulas. If the variables are correlated, then the complexity depends on the placement of the variables of the lineage in the junction tree i.e., if the lineage formula can be decomposed over the junction tree such that the subexpressions are present locally, the algorithm is efficient. In the worst case, when the variables are spread out arbitrarily, the algorithm can take time exponential in the size of the formula. This high variance in the processing time is troubling and must be mitigated.

**Redundant Variables:** Since our underlying data model is a *forest* of junc-

tion trees, there are a number of independence relationships that are present among the random variables. However, Algorithm 2 is currently unaware of these independence relationships and might perform unnecessary computation. We illustrate this with an example. Consider the index shown in Figure 6.2(iv). Suppose we are interested in to compute the probability of  $a \wedge o$ . The efficient way to process this lineage is to compute  $p(a)$  and  $p(o)$  separately since they are independent, and use them to determine the probability of  $a \wedge o$ . However, Algorithm 2 proceeds by making the following recursive calls on the child nodes  $I_1: \{a, c\}$ ,  $I_2: \{c, j\}$  and  $I_3: \{j, o\}$ , which is significantly more computation since we have to maintain joint probability distributions  $p(a, c)$ ,  $p(c, j)$  and  $p(j, o)$ . The reason behind this is that the knowledge of the disconnection is “hidden” in the leaf of the INDSEP and can only be discovered when the recursion reaches the leaf. Clearly, this computation is redundant since  $a$  and  $o$  are actually independent.

**Multiple Lineage formulas:** Many output tuples of a conjunctive query share common subexpressions in their lineage. Instead of computing the probability of the same expressions repeatedly, we can exploit this commonality by reusing the previously computed results. This could bring down computation time by a large fraction. We note here that such sharing is possible not only when the lineages share terms, it is quite useful even otherwise when the Steiner trees corresponding to the lineages share large paths. For instance, consider the lineages  $c \wedge n$  and  $b \wedge m$ . In this case, the lineage  $c \wedge n$  recursively generates ExpressionSets  $\{c, j\}$  on  $I_2$  and  $\{j, n\}$  on  $I_3$ . Similarly, the lineage  $b \wedge m$  generates ExpressionSets  $\{b, c\}$  on  $I_1$ ,  $\{c, j\}$  on  $I_2$  and  $\{j, n\}$  on  $I_3$ . The ExpressionSet  $\{c, j\}$  is common to both lineages and it needs to be computed

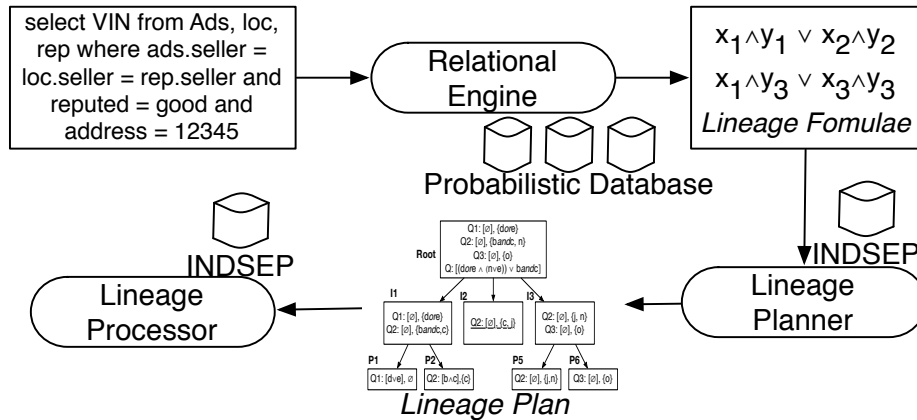


Figure 6.5: PrDB’s lineage processing component overview: Input conjunctive query is first executed by the relational engine which computes lineages of output tuples. Lineage Planner then computes an optimal plan for processing these lineages, which is executed by the Lineage Processor.

only once.

To effectively handle all the three issues discussed above, we introduce a lineage planning phase to our algorithm, which we describe in the next Section.

### 6.3 Lineage Planning & Evaluation

We use a two-pass approach to lineage processing - in the first step, we work through the index and formulate a plan for the lineage and in the next step, we execute the formulated lineage plan. We illustrate the complete sequence of query processing operations in Figure 6.5. As shown in the figure, a conjunctive query is first executed by a relational query processor and lineages of the output tuples are computed (Using query rewriting. See Chapter 3). Following this, the lineage planning and the lineage processing phases occur; we describe these here.

### 6.3.1 Lineage Planning

In this phase, we determine the lineage plan, i.e., the set of recursive calls to be made in each index node in the INDSEP data structure. In addition, we optimize the lineage plan by (a) identifying common subexpressions across a batch of lineages and sharing such computation. (b) identifying redundancies (c) determining the lwidth induced by lineage formula at the intermediate nodes in INDSEP.

**Naive Plan:** We first describe how to compute a naive lineage plan. Note that we are given a batch of lineages as input to the system. Just as in Algorithm 2, we determine the ExpressionSet corresponding to each child node. However, in this case, we have a list of ExpressionSets corresponding to each index node. We denote this list by the notation  $E^{node}$ . The expression set corresponding to the  $i$ th lineage is given by  $E_i^{node}$ . The lineage plan is a hierarchical data structure (corresponding to INDSEP) that essentially stores the  $E^{node}$  list against each node. Now we discuss how to optimize the naive lineage plan.

**Batch/Multiple Lineage Processing:** The INDSEP data structure naturally allows the sharing of computation between lineage formulas that share subformulas. This results not only in reduced number of disk accesses but also cpu processing time. Here, we look for duplicate ExpressionSets in each node in the lineage plan and remove them. After computing the naive lineage plan, each node  $N$  in the lineage plan stores the list  $E^N$  as described above. Now, we modify this list by removing the duplicate entries of ExpressionSets. This ensures that we will only execute distinct ExpressionSets. However, we need to maintain additional bookkeeping information corresponding to the duplication to execute the plan correctly. Specifically, we need a mapping from

the list  $E^P$  to the list  $E^N$  ( $P$  is the parent of  $N$ ). This mapping helps the lineage processor to correctly identify the parent recursive calls generating the ExpressionSets.

However, even more aggressive sharing can be performed. Suppose that processing lineage  $\lambda_1$  generates ExpressionSet  $\{j, m, n\}$  on child  $I_3$  and processing  $\lambda_2$  generates ExpressionSet  $\{m, n\}$  on child  $I_3$ . The above technique would treat the two ExpressionSets separately since they are different. However, a more useful technique here is to first compute  $p(j, m, n)$  by evaluating ExpressionSet  $\{j, m, n\}$  and then using this result to compute  $p(m, n)$  (by eliminating  $j$ ) which is the result of evaluating ExpressionSet  $\{m, n\}$ .

**Redundancy Detection:** For simplicity, we discuss the case of removing redundancies for a single lineage  $\lambda$  (The discussion extends to batch of lineages as well). We take care of detecting redundancies at the root level of the index itself. Given a lineage formula  $\lambda$  as input, we split it into multiple ExpressionSets, where each ExpressionSet corresponds to a connected component, just as we described in Section 6.1.3. We modify INDSEP to additionally store the knowledge of the components in the junction tree. For each random variable, INDSEP stores the id of the component to which it belongs in a hash table. The hash table is constructed while building the index and we use a Union-Find data structure [21] to maintain this data structure up-to-date in response to updates (inserting new tuple involves adding a new random variable). By splitting the input lineage in this manner, we guarantee that none of the nodes in the lineage plan contain an ExpressionSet with two independent variables. Hence, we never compute a joint pdf between a pair of independent variables. After splitting the lineage as described above, we use the previously described



multiple lineage planning algorithm to determine the lineage plan. In addition, we mark the root so as to *combine* the results of each of the lineages to produce the final result.

**Lwidth Computation:** After modifying the lineage plan as discussed above, we evaluate the feasibility of executing each step of the plan. The feasibility is determined by computing the lwidth value at each node in INDSEP, since we process ExpressionSets at each node. At the leaf nodes of the tree, which correspond to the disk partitions, we process ExpressionSets on the junction tree corresponding to the disk partition.(Step 20 in Algorithm 2). At each internal node of INDSEP, after building the IJT, we process ExpressionSets on it (Step 23 in Algorithm 2). To compute the lwidths, we use the eager+order heuristic described in Section 6.1. In addition, we also compute the optimal pivot locations. Note that we need not know the actual pdfs, but only the sets of variables over which they are defined. Hence, the time for computing lwidths and pivots is quite small, compared to the lineage processing times. When the junction tree is disconnected, we compute the lwidth and pivot for each partition separately along with the lwidth involved in combining the results from the different partitions together. We enforce an *lwidth threshold* on the computation in order to bound the lineage processing time. If the computed lwidth at a given node exceeds the threshold, then we mark the relevant node in the lineage plan to indicate that we need to perform approximations (Section 6.3.3). Our method ensures that we use approximations only for the portions of the lineage formula that have large lwidths and not for the complete formula as a whole. As we show in our experiments, this significantly improves the quality of our approximations.

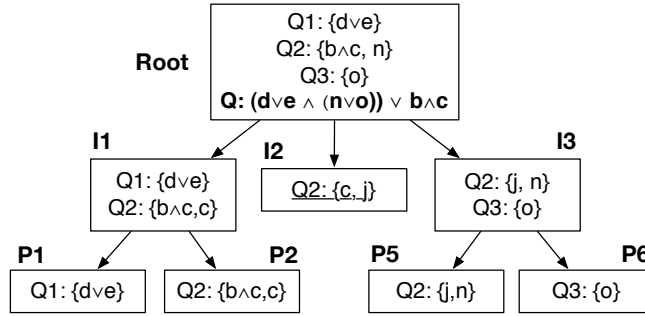


Figure 6.6: Lineage Plan for lineage  $\lambda = (d \vee e)(n \vee o) \vee (b \wedge c)$

### 6.3.2 Lineage Plan and Execution

**Lineage Plan:** As specified earlier, the lineage plan data structure specifies the list of ExpressionSets to be executed at each index node in INDSEP. The lineage plan is a tree based data structure, where each node in the tree corresponds to one of the index nodes in INDSEP. Each lineage plan node  $N$  (with parent  $P$ ) in the lineage plan contains the following which were computed in the previous section:

- (C1) Id of the index node to which it corresponds
- (C2) List of ExpressionSets:  $E^N = \{E_1^N, E_2^N, \dots\}$
- (C3) Optimal pivot(s), lwidth(s) - do we approximate ?
- (C4) Can we get results from shortcut potentials ?
- (C5) Pointers to children, Hashtable ( $E^P \rightarrow E^N$ )
- (C6) Whether to combine multiple lineage results ? (e.g., due to disconnections)

An example of a simple lineage plan, for the lineage  $\lambda = (d \vee e)(n \vee o) \vee (b \wedge c)$  is shown in Figure 6.6. Notice that we have indicated the INDSEP node to

which each plan node corresponds. Owing to disconnections,  $\lambda$  is initially split into three ExpressionSets  $\{d \vee e\}$ ,  $\{b \wedge c, n\}$  and  $\{o\}$ . Hence, the root has 3 ExpressionSets, and an additional ExpressionSet while tells the lineage processor to combine the results together.  $Q_1$  introduces recursive calls  $\{d \vee e\}$  over  $I_1$  as shown in the figure. Also  $Q_2$  introduces recursive calls  $\{b \wedge c, c\}$ , on  $I_1$ ,  $\{c, j\}$  on  $I_2$  and  $\{j, n\}$  on  $I_3$  as shown. Note that the ExpressionSet corresponding to  $I_2$  is marked, since it can be directly obtained from the shortcut potential of  $I_2$ .  $Q_3$  introduces recursion over  $I_3$  as shown in the figure.

**Executing the Plan:** We execute the lineage plan via recursion over the lineage plan structure. We explain the key recursive step here. Given a lineage plan  $P$  on an index node  $I$ , we recursively assign children of  $P$  to the respective child nodes of  $I$ . We collect the results of the executions of each of the child nodes and construct the IJTs (Section 6.2) using the hashtable mappings. Now, we execute the ExpressionSets contained in  $I$  over the IJTs using the optimal pivot locations, and return the result to  $I$ 's parent. We have 2 special cases to take care of: (1) when the lineage plan is marked, we directly obtain the result from the shortcut potential (2) Whenever the lwidth exceeds the threshold, we perform approximations while processing lineage (Section 6.3.3). Whenever the index node corresponds to a leaf, we execute the ExpressionSets on the junction trees corresponding to the leaf and return the results to the parent node.

### 6.3.3 Approximation Technique

In this section, we describe how we deal with lineage formulas that induce large lwidths on the underlying junction tree. Currently, we use a simple Monte Carlo technique which is based on Gibbs sampling [43]. The accuracy of the estimates can be improved by using more samples. In our ongoing work we are developing new techniques based on modifying lineage formula to allow efficient processing, similar to the approximate lineage computation of Re et al. [92].

The central idea behind this technique is to use samples of the probability distributions and pass them around as messages, instead of the complete pdfs. We modify the Eager and the Eager+Order message passing algorithms of Section 6.1 in order to support sampling techniques.

The algorithm we present is a recursive algorithm on the junction tree. We illustrate it with a simple example. Suppose that we want to compute the probability of  $(a \wedge k) \vee h$  in the Junction tree of Figure 6.2(iii). We construct the Steiner tree corresponding to  $a$ ,  $k$  and  $h$  as before and also select a pivot node. Suppose we select clique (ac) as pivot. In the first step, the clique (ac) constructs  $N$  samples from the pdf  $p(a, c)$  and sends it to clique (cfg). The clique (cfg) now computes  $p(g, f|c)$ , (dividing  $p(g, f, c)$  by  $p(c)$ ) and uses the samples  $(c_i)$  from (ac) to generate samples  $(g_i, f_i)$  from  $p(g, f|c)$ . Now clique (cfg) recursively sends samples  $g_i$  to clique (gjk) and samples  $f_i$  to clique (fh). These cliques, return samples corresponding to  $k$  and  $h$  respectively to clique (cfg). Now (cfg) combines these samples and returns them  $(c_i, h_i, k_i)$  to clique (ac), which now evaluates the probability of  $(a \wedge k) \vee h$ .

## 6.4 Experimental Evaluation

### 6.4.1 Implementation Details

We implemented the lineage processing component of the query processor module in PrDB (Figure 3.1) for our experimental analysis. As discussed in Section 3.4, we use a query rewriting approach to construct the lineages of the output tuples. Subsequently, we use INDSEP to compute the probability of the lineages. We note here that lineages computed by the query rewriting may not be in the *read-once* format. Hence, we implement the co-graph recognition algorithm of Golumbic et al. [45] to rewrite a boolean formula as a read-once function. This algorithm is known to be complete, i.e., if the boolean formula has a read-once form, then it will find one such representation.

### 6.4.2 Experimental Setup

The main objectives of our experimental analysis are to show the benefits of (1) our heuristics for processing lineage formulas over junction trees (2) our lineage planning algorithms to improve lineage processing times (3) our approximation algorithms to generate accurate results. We begin with a discussion of the experimental setup.

**Dataset:** We generated a synthetic dataset with 3 relations  $R_1, R_2, R_3$ , each of size 100,000 tuples that correspond to the Car Ads application (Section 1.1.2). All tuples are uncertain. In addition, each tuple in the relation was correlated with a random number (between 2-10) of other tuples in the database. The correlations are randomly generated factors that correspond to the conditional probability distributions. After populating the database, we build the junction

forest (set of junction trees) and use the algorithms from [62] to construct the INDSEP data structure. To generate databases with varying amounts of correlations, we vary the sizes of the connected partitions in the junction tree. We generate 3 different datasets, each with different partitioning sizes. Database  $D_1$  has single node partitions, i.e., it is a tuple independent database.  $D_2$  has partitions of size 100, i.e., it is lightly correlated.  $D_3$  is a *moderately correlated* database with partitions of size 1000. In addition, we generate a Markovian sequence denoted by  $M$  [61], which is a single junction tree.

**Query Workload:** We used the query  $Q$  from the introduction as part of our experiments. We carefully vary the data in the join columns in order to create lineages of different sizes. In addition, we also generate artificial lineage boolean formulas of the form  $A_1 \wedge A_2 \wedge A_3$  as required to illustrate the results. Each of the  $A_i$ 's are disjunctions of the tuple uncertain random variables.

Our most important experimental findings are as follows:

**Suitability of INDSEP for Lineage queries:** In this experiment, we evaluate the benefits of INDSEP for processing lineage queries. As noted before, INDSEP is extremely useful for *inference* and *aggregation queries*. We run the lineages of various sizes on the Markovian sequence database  $M$  alternatively, using (a) INDSEP (b) directly on the underlying junction tree. We measured the wall-clock times for processing the lineage formulas for both cases. The bar graph containing these results is shown in Figure 6.7(a). The results are plotted as a function of the size of the formula. As noted before, we observe exponential decrease in the amount of time required to process lineage formulas. Note that the y-axis is in *log* scale, so the benefits of INDSEP are more substantial than apparent.

**Performance of the heuristics:** We now evaluate the performance of our heuristics Eager and Eager+Order for evaluating boolean formulas over a junction tree as opposed to the Naive approach. We used both the heuristics alternatively to bottom out the recursion (Section 6.2). We used both *independent* database  $D_1$  and the *lightly correlated* database  $D_2$  for this experiment. We compare the time taken to evaluate a lineage formula for both the heuristics as a function of the size of the formula. In addition, we also compared the above heuristics with the Naive approach (Section 6.1). We performed the experiment for lineages of different sizes varying from very small 10 to very large lineages 150. The time taken for lineage processing is plotted in Figures 6.7(b) & (c). Notice that the y-axis is in log-scale. As shown in the figures, the Naive and the Eager approaches perform very poorly as compared to our Eager+Order heuristic. Even for small lineages below 30, the amount of processing time is very large, since very large intermediate pdfs are created by the heuristic. In contrast, even for large lineage formula, the heuristic Eager+Order performs very efficiently as shown in the figure (taking less than 0.4 seconds for lineage of size 30 and about 7 seconds for lineage of size 100).

**Study of Lineage Processing Performance:** With this experiment, we illustrate the performance of our lineage processing algorithm as a function of the size of lineage. For each of the databases  $D_1$ ,  $D_2$  and  $D_3$ , we evaluate the time taken for exact lineage processing (we set the lwidth threshold to be  $\infty$ ) for different lineage sizes. The times are plotted in Figure 6.7(c). As shown in the figure, the lineage processing time increases as the database becomes more correlated owing to the larger intermediate pdfs generated during lineage processing. As we can see from the figure, once the lwidths generated exceed 20,

(the intermediate pdfs =  $8MB$ ) the algorithm is largely unusable. The exponential blow-up due to the large intermediate pdfs is evident from the figure. Hence we introduce the lineage planning phase in our approach and introduce the lwidth threshold parameter to characterize such cases and resort to approximation techniques for them. In addition, the figure also shows the time taken by a special purpose technique such as Mystiq [27] and Sen et al. [96] to process lineages on the dataset  $D_1$ . As shown earlier (Section 6.1.3), our algorithm is quadratic in the size of the formula, while special purpose techniques are linear. Not surprisingly, our system performs poorly with respect to a special purpose technique for tuple independent probabilistic databases.

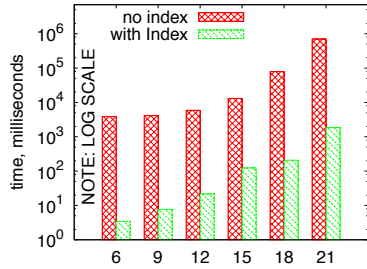
**Lineage Planning (Batch Lineages):** With this experiment, we show the benefits of processing multiple lineages together using our batch processing algorithm (Section 6.3.1). We use correlated database  $D_2$  for our experiments and we measure the time taken to process a workload using the batch lineage processing algorithm and for comparison, we process each lineage in the workload individually. We perform the comparison for both small lineages ( $< 20$ ) and large lineages ( $> 50$ ) which were chosen randomly. The results are plotted in Figure 6.7(e). As shown in the figure, the time taken by the batch lineage processing algorithm is less than the time taken for processing each lineage individually and add them up. Notice that even for randomly generated lineages without any explicit sharing between the formulas, we obtain significant reduction in the lineage processing time. In addition, as the workload sizes get very large  $> 1000$ , we gradually lose the benefits of the batch lineage processing. This is because of the overheads that arise in the lineage planning, i.e., removing duplicates. We now evaluate the performance of the batch lineage



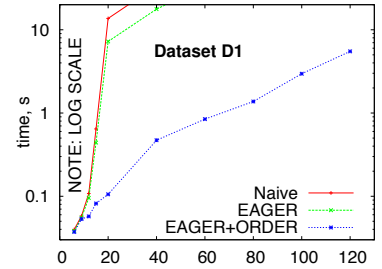
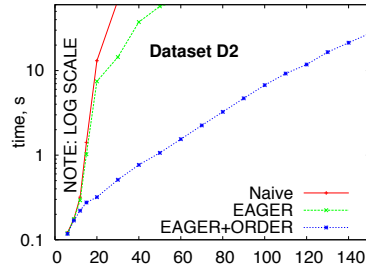
processing algorithm as a function of the amount of sharing that is present between the lineages. This way, we create workloads with a sharing coefficient of 0.0 (no overlapping variables) to 0.6 (60% repetitions of variables). We now evaluate these workloads on both datasets  $D_1$  and  $D_2$ . We plot the ratio of the time taken for batch lineage processing to the time taken for processing each lineage separately (Smaller ratios are better). As shown in Figure 6.7(f), for both datasets  $D_1$  and  $D_2$ , we can see that the lineage processing time reduces as the sharing coefficient increases. Also, we find that the batch lineage processing is more beneficial for correlated datasets than for completely independent datasets. In fact, it induces an overhead for  $D_1$  at low sharing coefficients. This occurs due to the blowup of the workload size at the beginning of the algorithm where we split the lineage into multiple ExpressionSets for each component in the junction forest (the splitting is biggest for  $D_1$  since it is disconnected). We also found that the batch processing algorithms is quite beneficial for sharing across inference queries.

**Approximate Lineage Processing:** Now, we evaluate the accuracy of our approximate lineage processing algorithms. We use the Markovian sequence dataset  $M$ , i.e., the fully connected junction tree. We first compute the error in the output probability as a function of the number of samples used in our Monte Carlo algorithm. We compute the exact probabilities by setting the lwidth threshold to  $\infty$ . Since we were using the fully connected junction tree, we had to limit the size of the formula to less than 20. We used three workloads depending on the size of formula - (5-10), (10-15) and (15-20). The results are shown in Figure 6.7(h). As shown in the figure, for each of the three workloads, we observe that the accuracy rate improves with the

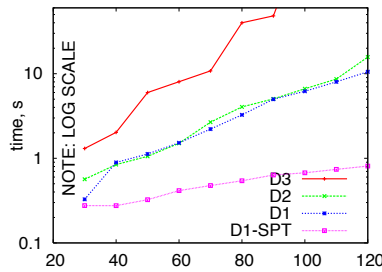
number of samples. In fact, when we use more than 500 samples, we only notice a difference in the second decimal position. With 2000 samples, we notice an error in the third decimal position. Next, we illustrate the benefit of using INDSEP for our approximation. In this experiment, we carry out the approximation algorithm, first directly on the underlying junction tree and then using INDSEP(1000 samples). We used three kinds of lineage formula, short range, medium range and long range lineages - which have varying spans on the underlying junction tree. Span of a lineage is the size of the Steiner tree induced by the lineage formula. As shown in Figure 6.7(i), the amount of error using INDSEP is smaller than the error without it. This is due to 2 reasons: First, this is because of selectively approximating only portions of the formula that lead to large lwidth. Second, due to the shortcut potentials in INDSEP, the size of the effective junction tree is small. Also, the difference in the errors is more pronounced for long range lineages. This is because the errors continuously add up sequentially for large span lineages when we process the lineage without INDSEP.



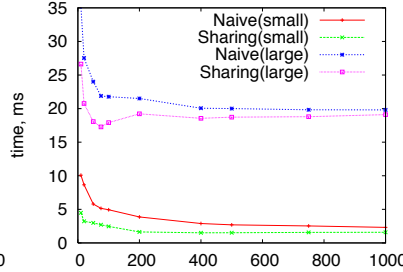
(a) Time vs Lineage size



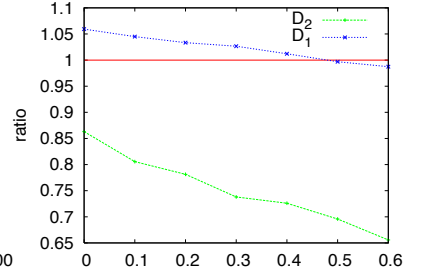
(b) and (c) Heuristic performance vs Lineage Size



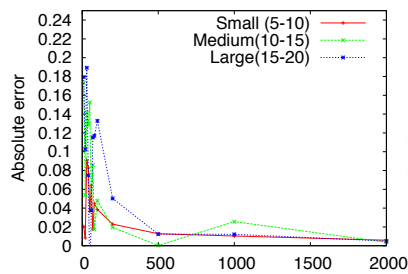
(d) Time vs Lineage size



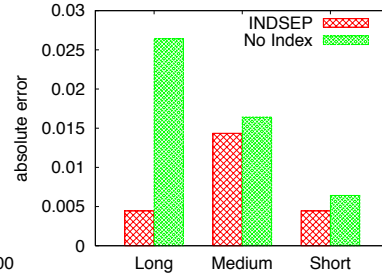
(e) Time vs Workload size



(f) Ratio vs sharing factor



(g) Error vs Sample size



(h) Error vs Range of lineage

Figure 6.7: Results: (a) Processing lineages using INDSEP is more scalable. (b) and (c) Illustrating benefits of EAGER+ORDER heuristic over the EAGER and naive approaches for datasets  $D_2$  and  $D_1$ . (d) As the correlations increase, lineage processing times increase. Special purpose technique (SPT) performs better for the independent dataset  $D_1$ . (e) and (f) Illustrating benefit of batch lineage processing. (g) Sampling errors can be reduced by increasing number of samples. (h) INDSEP improves the quality of our approximations significantly.

## Chapter 7

# Query Processing on Markovian Sequences

So far in the dissertation, we have focused on representing general probabilistic databases and developing efficient algorithms for query processing over them. However, uncertainty is also highly prevalent in *data streams*. As discussed in Chapter 1, probabilistic modeling of large-scale data generated by sensor networks and other measurement infrastructure is one of the primary applications that generates uncertain probabilistic streams. In this chapter, we address the problem of efficient query evaluation over such probabilistic streams. We observe that although probabilistic streams tend to be strongly correlated in space and time, the correlations are usually quite *structured*, with the same set of dependences and independences repeated across time. Furthermore, most real-world probabilistic streams are *Markovian* in nature, with the state at time “t+1” being independent of the states at previous times given the state at time “t” (in some cases, the state at time “t+1” may depend on a fixed number of states in the recent past [20]). Now, we describe

how to represent these sequences efficiently and design algorithms for query processing over these sequences. The outline of the chapter is as follows. In Section 7.1, we formally define a *Markovian* sequence/stream. Following this, we develop an algebra of operators on Markovian streams in Section 7.2 and develop algorithms for them in Section 7.3. In Section 7.4 we discuss query planning algorithms and the associated optimizations. We conclude with an experimental evaluation of the stream processing component of the PrDB system.

## 7.1 Markovian Sequences

Although several probabilistic streams that occur in real world applications are highly correlated in both space and time, we observe that the correlations are very structured with the same set of dependences and independences repeated across time. Further, the correlations are *Markovian*, which means that the tuple at time  $t + 1$  is independent of the tuple at time  $t - 1$  if we know the value of the tuple at time  $t$ . We can store such Markovian sequences very efficiently by treating the correlation structure of the PGM as a schema and decoupling it from the probability distribution numbers. We formally illustrate it below.

**Definition 1.** A Markovian Sequence,  $S^p = \mathbf{S}^1, \mathbf{S}^2, \dots$ , is a probabilistic sequence that satisfies the Markov property, i.e., the set of random variables  $\mathbf{S}^{t+1}$  is conditionally independent of  $\mathbf{S}^{t-1}$  given the values of the random variables in  $\mathbf{S}^t$  (denoted  $\mathbf{S}^{t-1} \perp\!\!\!\perp \mathbf{S}^{t+1} | \mathbf{S}^t$ ).

Because it obeys the Markov property, any Markovian sequence is completely determined by the joint probability distributions between successive

sets of random variables,  $p(\mathbf{S}^t, \mathbf{S}^{t+1}), \forall t$ . Therefore, we can represent a Markovian sequence as a sequence of joint probability distributions.  $p(\mathbf{S}^1, \mathbf{S}^2), p(\mathbf{S}^2, \mathbf{S}^3), \dots, p(\mathbf{S}^t, \mathbf{S}^{t+1})$ . The repeating structure in the Markovian sequences can be captured using a combination of two components:

- The first component, called the *schema graph*, is the PGM representation of the two step joint distribution that repeats continuously throughout the sequence.
- The second component, called *clique list*, is the set of *direct* dependencies that are present in the sequence between two successive sets of random variables.

The schema graph and the clique list of the example Markovian sequence discussed above are shown in Figures 7.1(b,c).

This repeating structure also allows us to compactly represent a Markovian sequence as a sequence of tuples, each of which is an ordered list of CPDs corresponding to the clique list. Furthermore, since the CPDs for each time instance, and the domains of the random variables are known in advance, we can also remove the schema information and simply transmit the numbers comprising the CPDs.

For the example shown above, the list of CPDs at time  $t$  is:

$\{p(X^t), p(Y^t|X^t), p(X^{t+1}|X^t), p(Y^{t+1}|X^{t+1}, Y^t)\}$ . Assuming all variables are binary, we instead represent these CPDs as an array of numbers:  $\{p(X^t = 0), p(X^t = 1), p(Y^t = 0|X^t = 0), p(Y^t = 1|X^t = 0), \dots, p(Y^{t+1} = 1|X^{t+1} = 1, Y^t = 1)\}$  (total 18 numbers). This allows us to efficiently transfer the tuples between the operators, and minimize the memory requirements of our operators.

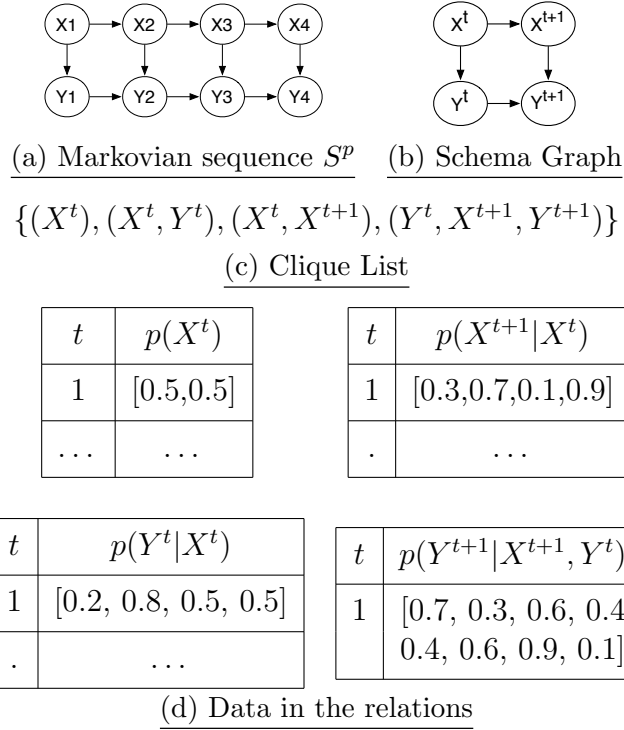


Figure 7.1: (a) Example of a Markovian sequence  $S^p$  on attributes  $X$  and  $Y$ ; (b) Schema graph and (c) clique list of  $S^p$ ; (d) Representing  $S^p$  using one relation per CPD.

## 7.2 Probabilistic Sequence Algebra

The probabilistic sequence algebra underlying our system is a probabilistic extension of the sequence algebra model proposed by Seshadri et al. [100] (with some minor changes). The result of the application of an operator on a probabilistic sequence is equivalent to applying the operator to each of the possible sequences separately, and then adding the result sequences to a *result set*. If two sequences return the same result, then we just add up the probabilities of the sequences together. The *result set* is also a set of possible sequences and is therefore a probabilistic sequence. Formally, applying operator  $op$  to prob-

abilistic sequence  $S^p$  results in a probabilistic sequence  $R^p = op(S^p)$  where,

$$Prob(R^p = x) = \sum_{PS_i \in S^p | op(PS_i) = x} p_i$$

We use this definition to extend the sequence algebra operators such as *project*, *set union*, *aggregates* to probabilistic sequences. However, two of the operators deserve further discussion:

- **Selection:** The selection operator for a sequence is different from relational selection because we cannot drop tuples in our deterministic sequence model [100]. If we drop tuples, then the sequence loses the property that the  $t^{th}$  tuple corresponds to the set  $\mathbf{S}^t$ . If a tuple in a deterministic sequence does not satisfy a predicate, rather than deleting the tuple, we make note that tuple does not exist by creating a new binary valued attribute  $A_P^t$ , where  $P$  is the selection predicate.  $A_P^t$  is assigned a value 1 if the tuple  $\mathbf{S}^t$  satisfies the predicate and 0 otherwise. If the selection predicate is over a probabilistic attribute, then  $A_P^t$  itself would be a probabilistic attribute. We discuss this further when we present our operator algorithms.
- **Join:** We currently restrict our implementation to equi-joins on time. To join two probabilistic sequences,  $S^p$  and  $T^p$ , we compute the results of join between every possible sequence  $PS_i$  of  $S^p$  and every possible sequence  $PT_j$  of  $T^p$ ; the probability of the result is the product of the probabilities of  $PS_i$  and  $PT_j$ .

Along with the standard sequence operators, we introduce two new operators specific to probabilistic streams. Both these operators take probabilistic sequences as input and return deterministic sequences as output.



1. **MAP:** The MAP operator returns the sequence in the set of possible sequences that has the highest probability.

$$MAP(S^p) = \{PS_i \in S^p | \forall PS_j \in S^p, p(PS_i) \geq p(PS_j)\}$$

2. **ML:** The ML operator constructs a new deterministic sequence whose  $t^{th}$  tuple is the most likely  $t^{th}$  tuple over all the possible sequences. Suppose we denote the  $t^{th}$  tuple in the deterministic sequence  $D$  by  $D^t$ . Then, formally,  $ML(S^p) = D$ , where:

$$D^t = \mathbf{argmax}_{x \in X_t} f^t(x),$$

$$\text{where } X_t = \bigcup_{PS_i \in S^p} PS_i^t \text{ and } f^t(x) = \sum_{i | PS_i^t = x} p_i$$

We note that the selection operator commutes with both the MAP and ML operators. Similarly, the join operator commutes with both selection and projection operators. We prove these results in our technical report [58]. These properties help us in designing more efficient query plans for queries. We also notice that in general, the projection operator does not commute with the MAP and ML operators. However, for the restricted case of *Markovian sequences* which we describe next, we can still establish the commutativity of the projection operator with the aggregation and the windowing operators, which is very crucial for query optimization.

### Operating on Markovian Sequences

Since Markovian sequences are a special case of probabilistic sequences, the operators defined in Section 7.2 can be used to operate upon Markovian sequences. However, Markovian sequences are not closed under that set of operators. Several of the operators take Markovian sequences as input and return non-Markovian sequences as output depending on the input schema. We formalize this observation by defining the notion of a *safe operator-input* pair.

**Definition 2.** A safe operator-input pair is a combination of an operator and an input sequence schema such that the operator returns a Markovian sequence as output when applied to a Markovian sequence with the specified input schema.

Identifying safe operator-input pairs is crucial because we can evaluate such operators very efficiently; on the other hand, if an operator is not safe for an input schema, then we may have to resort to approximation schemes. Safe operator-input pairs can also be chained together with other safe pairs in a sequence to form polynomial-time query plans for complex queries. In our query optimization framework (Section 7.4.2), we design an operator ordering algorithm that avoids non-safe operators as much as possible.

We remark that despite the similarity of this concept to *safe plans* [27], there are several differences between the two concepts. Safe plans were developed for probabilistic databases with only independent tuple-level uncertainty; whereas Markovian sequences exhibit high degrees of correlations. Further, safe plans require global reasoning over the database schema and the query, whereas safe operator-input pairs are defined locally without any global consideration. We also note that query processing over Markov sequences is intractable even if we don't allow joins, in contrast to independent tuple databases where single relation queries are trivially answerable.

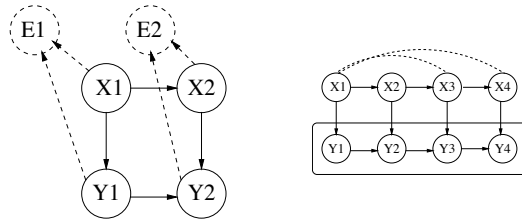
### 7.3 Operator Algorithms

In this section, we present detailed description of our algorithms for operating on Markovian sequences in accordance with the semantics defined in the previous section. Our system also supports *sliding window* variants of the

aggregate operators, a *pattern* operator that identifies user specified patterns in the stream, and we present the details of that as well. Our operators are designed to be incremental and treat the Markovian sequence tuples as a data stream, operating on one tuple (corresponding to a joint distribution between the variables at two consecutive positions in the Markovian sequence) at a time. If the operator-input pair is safe, then the output is also produced in the same fashion (a tuple at a time). If an operator-input pair is not safe, then we resort to approximations.

Each operator that we have designed implements two high-level routines: (1) a *schema routine*, which is invoked when the operator is instantiated, examines the schema of the input sequence and deduces the schema of the output sequence (to be fed to the input of the next operator); (2) a *get\_next() routine* that is invoked every time a tuple is routed through this operator. We describe only the high level details of the algorithms here, a comprehensive description can be found in our technical report [58].

**Selection** In the schema routine, we first start with the PGM corresponding to the input schema. Then we add a new node corresponding to the *exists* variable ( $A_P$ ) to both time steps of the PGM. We connect this node to the variables that are part of the selection predicate through directed edges. In addition, we update the clique list of the schema to include the newly created dependencies. An illustration of this operation is shown in Figure 7.2(a). Here, we have as input, the Markovian sequence  $S^p$  shown in Figure 7.1(a), and a predicate  $X > Y$ . We add a new node  $E$  corresponding to the new exists variable, and directed edges from  $X$  and  $Y$  to  $E$ . Also, we add  $(E^t, X^t, Y^t)$  to the clique list. In the *get\_next()* routine, we determine the CPD of the newly



(a) Selection predicate ( $X > Y$ ) (b) Projection is unsafe

Figure 7.2: (a) Executing a selection predicate ( $X > Y$ ) entails adding new *exists* variables ( $E^i$ ); the dotted edges show the changes to the schema. (b) Projection may result in a non-Markovian sequence – if  $Y$  nodes are eliminated, the resulting  $X$  sequence (shown through dotted edges) is not Markov.

created node, add it to the input tuple’s CPD list and return the new tuple. A typical example of a CPD for such a case (predicate:  $X > Y$ ) is shown in Figure 7.3(a).

As we can see, the algorithm does not alter the Markovian property of the sequence, and therefore every sequence can be paired safely with this operator.

### Projection

In this operator, we need to remove the nodes that are not in the projection list. This corresponds to an *elimination* operation on the graphical model. To determine the schema of the output sequence, we need to determine if any new edges need to be added to the schema graph (as a result of the elimination). We do this by performing a dummy inference operation on the input schema graph and determine the new edges to be added to the graphical model. We then derive the output sequence schema from the graphical model. In the `get_next()` routine, we perform the actual variable elimination procedure to eliminate the nodes that are not required.

The projection operator is *not* always *safe* for all input sequences. In certain cases, even if the input is a Markovian sequence, after projection, the output

$X^i Y^i E^i$	$f$	$G^1 X^2 G^2$	$f$	$G^1 E^2 X^2 G^2$	$f$
0 0 0	1	0 0 0	1	0 0 1 0	1
0 0 1	0	0 0 1	0	0 1 1 0	0
1 0 1	1	0 1 0	0	. . . .	.
1 0 0	0	0 1 1	1	1 0 1 2	0
0 1 0	1	1 0 0	0	1 0 1 1	1
0 1 1	0	1 0 1	1	1 1 1 2	1
1 1 0	1	. . .	.	1 1 1 1	0
1 1 1	0	1 1 2	1	. . . .	.

(a)  $X > Y$     (b)  $G^2 = X^2 + G^1$     (c)  $G^2 = X^2.E^2 + G^1$

Figure 7.3: Constructing CPDs for new nodes for (a) selection, (b) aggregate, and (c) aggregate with selection ( $dom(X) = dom(Y) = \{0, 1\}$ ).

may not be a Markovian sequence. An example of such a sequence is shown in Figure 7.2(b). Here, if the nodes denoted by  $Y^1, Y^2 \dots Y^n$  are eliminated from the sequence (i.e., if  $Y$  is removed), edges will be introduced between every pair of nodes  $(X^i, X^j)$  in the graph, which results in a non-Markovian sequence.

We characterize the schema of the input probabilistic sequence which results in unsafe projection as follows. Consider the connected subgraph  $G$  of the schema that contains the set of nodes  $E$  being eliminated. If there is an edge between the set of nodes  $E^t$  and  $E^{t+1}$  and there exists node  $x \in vertices(G) \setminus E$ , then after projection, the resulting sequence will not be Markov and the projection operation is unsafe. In Section 7.4.2, we present an algorithm to identify such scenarios and to postpone the projection operation if it is unsafe.

### Joins

In the schema routine, we concatenate the schemas of the two sequences in order to determine the resulting output schema, i.e., we combine the schema graphs, and concatenate the clique lists. Similarly, in the `get_next()` routine, we concatenate the CPD lists of the two tuples, whenever both tuples have the

same time value. Thus, a join can be computed incrementally, and is always safe.

### Aggregation

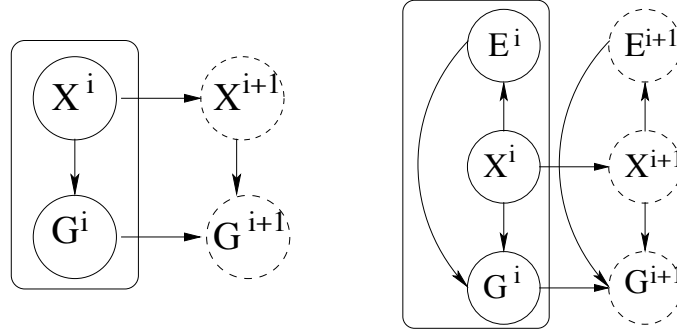
We support decomposable aggregates, SUM, AVG, MAX, MIN, COUNT, in our system. When computing aggregates, the output schema is just a single attribute corresponding to the aggregate (note that these are *not* sliding window aggregates, but rather one-time aggregates). The more complicated routine is the `get_next()` routine for which we have developed incremental algorithms. We consider two cases based on the query, the first when there are no selection predicates in the query, the second when there are selection predicates. We illustrate both cases with examples.

#### Case 1: No selection predicates

Consider a single attribute Markovian sequence  $X^1, X^2, X^3 \dots$ , and say we wish to determine the SUM of all the variables in the sequence, in an online fashion. Let  $G_k$  denote the sum of all the  $X^i$ 's from 1 to  $k$ . The trick we use here, is to incrementally compute the distribution  $p(X^i, G^i)$  as input tuples arrive.  $p(X^{i+1}, G^{i+1})$  can be incrementally computed from  $p(X^i, G^i)$  as follows:

$$p(X^{i+1}, G^{i+1}) = \sum_{X^i, G^i} p(X^i, G^i) p(X^{i+1} | X^i) p(G^{i+1} | G^i, X^{i+1})$$

At the end of the sequence, we get  $p(X^n, G^n)$ , from which we can obtain  $p(G^n)$  by eliminating  $X^n$ . The PGM corresponding to this operator is shown in Figure 7.4 (a). The CPD  $p(G^{i+1} | G^i, X^{i+1})$  is determined based on the nature of the aggregate (Figure 7.3(b) shows a SUM CPD).



(a) Case 1 (no selection predicates) (b) Case 2 (with selection predicates)

Figure 7.4: Illustrating aggregate computation.  $G^i = \text{Agg}(X^1, X^2, \dots, X^i)$ . In each `get_next()` call, dotted variables are added to the PGM, and the boxed nodes are eliminated, continuously maintaining state  $p(X^i, G^i)$  and  $p(X^i, G^i, E^i)$  respectively. Also, note the dependence of  $G^i$  on  $E^i$  in (b).

### Case 2: With selection predicates

When selection predicates are present, the PGM that we construct is slightly more complex. An example is shown in Figure 7.4(b). This is because of the presence of the  $E^i$  (exists) attributes: a value  $X^i$  contributes to the aggregate only if  $E^i$  is 1 and not otherwise. This information is added to the CPD of the aggregate node, an example of which is shown in Figure 7.3(c). In this case, we maintain the distribution  $p(X^i, G^i, E^i)$  for all time instants  $i$  and determine the  $p(X^{i+1}, G^{i+1}, E^{i+1})$  from  $p(X^i, G^i, E^i)$  using a similar operation as described earlier. The PGM for doing this is shown in Figure 7.4(b).

In general, we have to maintain the distribution of all random variables in one time instance to enable incremental computation of aggregates. For **AVG**, we maintain the joint distribution of **SUM** and the **COUNT** aggregates for each time instant, and determine the distribution of **AVG** based on this.

The time complexity of the aggregate operator is  $O(D^3)$  for **MIN** and **MAX** aggregates and  $O(nD^3)$  for **SUM**, **COUNT** and **AVG** aggregates, where  $D = |\text{dom}(X^i)|$  and  $n$  is the length of the sequence. This is because the domains of the  $G^i$

variables for `SUM` and `COUNT` increase as  $i$  increases ( $|dom(G^n)| = nD$ ). Hence the CPD sizes increase resulting in high per tuple processing time as we receive more and more tuples. In order to keep the per tuple processing time for `SUM` and `COUNT` small, we use constant-time approximation algorithms for domains larger than a threshold parameter. We discuss these strategies in Section 7.4.3.

In addition, we also support entity based aggregates [16], for example, to determine the time instances at which a variable value was maximized. Details of the entity aggregate operator can be found in [58].

### Sliding Window Aggregates

A sliding window aggregate query asks to compute the aggregate values over a window that shifts over the stream. It is characterized by the *length* of the window, the desired *shift*, and the type of aggregate. Sliding window operator is unsafe for all input sequences, since the output of the operator is always non-Markovian (illustrated in Figure 7.5(a) and (b), formal proof in [58]). This is because the aggregate value for a sliding window influences the aggregates for all of the future windows in the stream. Therefore, the exact answer to the sliding window aggregate query has exponential data complexity, which forces us to use approximations.

One approach to handling this, that we adopt, is to ignore the dependencies between the aggregate values produced at different time instances, and to compute the distribution over each aggregate value independently. We achieve this by splitting the sliding window PGM into separate graphical models (one for each window), run inference on each of them separately and compute the results. Figure 7.5(c) shows a simple illustration of the operation that computes the marginal probability distributions of each of the nodes  $G^1$ ,  $G^2$ ,  $G^3$ ,



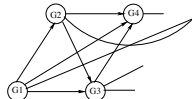
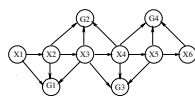
$G^4$ . The unmarked nodes in the figure denote the intermediate sums (we have used the decomposability property of our aggregates here).

However, for the special case of *tumbling windows*, where the length of the sliding window is equal to its shift, we can compute exact answers in a number of cases. We use a similar trick that we used for aggregates, i.e., we maintain the distribution of all the random variables in the last step of each window. By doing so, we can guarantee that the output sequence is Markovian. However, this still requires a final unsafe projection operation; we postpone that for as long as possible, and resort to approximation when the projection must be done. As shown in Figure 7.5(d), we eliminate only the boxed nodes and end up with a Markovian sequence with schema shown in Figure 7.5(e). Eliminating  $X^3$  and  $X^6$  is postponed to a later projection step.

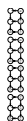
### **Pattern operator**

A *pattern* is a list of predicates on the attribute values, with the ordering of the predicates defining a temporal order on the sequence. For instance,  $(A = 3, B > 5, A < 3)$  is a pattern that looks for a sequence of time instants such that the value of attribute  $A$  is 3 in the first instant, the next  $B$  has value more than 5, and the following  $A$  has value less than 3. We currently only handle consecutive patterns. To compute the probability of a consecutive pattern, we need to compute the product of the corresponding conditional distribution functions. If the user specifies a threshold parameter, we can prune out those time steps that do not contribute to the result. For instance if we want a pattern with probability greater than 0.7, then each of the contributing CPDs must be at least 0.7.

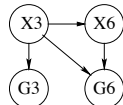
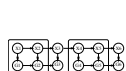
### **MAP operator**



(a) Sliding window model (b) Exact sliding window



(c) Approximate Sliding Window



(d) Boxed nodes are eliminated (e) Output

Figure 7.5: (a) PGM for sliding window aggregate.  $G^i$ 's denote the aggregates that we have to compute. (b) After eliminating the  $X^i$  variables, we obtain a clique on the  $G^i$  variables, which is  $\#P$ -hard. (c) Hence, we split the PGM into components as shown. Unmarked nodes are intermediate aggregates. (d) For tumbling window aggregates, we only eliminate boxed nodes to obtain the Markovian sequence shown in (e). Removing nodes  $X^3$  and  $X^6$  is postponed to a later projection.

The MAP operator takes in a Markovian sequence and returns a deterministic sequence. It is usually the last operator in the query plan, and hence it does not have a schema routine. The `get_next()` routine uses the dynamic programming based approach of Viterbi's algorithm [88] on Markovian sequences to determine the sequence that has the maximum probability. We have designed and implemented an incremental version of this algorithm by maintaining appropriate state in memory. For each value in the domain, we maintain the best sequence that ends in that value. After receiving the CPD list of the new

tuple, we extend each of the sequences that we have maintained in memory, by concatenating one additional value to it and computing its probability. After this, we update our memory state by recomputing the best sequences. We store these sequences in memory using a circular list of finite size. When the size of the sequence exceeds the length of the list, we remove the head of the list and continue our algorithm using the part of the sequence present in the list.

### **Most Likely operator**

In order to determine the most likely value of a variable at each time step, we first compute the marginal probability distribution for each time instant from each tuple. Based on this, we eliminate the variables that are not required and determine the most likely values for the variables. Further details can be found in [58].

## **7.4 Query Evaluation**

We begin with a brief discussion of our query language, and then present our overall query processing and optimization algorithms for evaluating queries over Markovian sequences.

### **7.4.1 Query Syntax**

In our system, queries can be specified either in an SQL-style language or it can be specified using the probabilistic sequence algebra described in Section 7.2. The SQL-style syntax is as follows:

```
<SELECT-MAP/ML> <Agg<attrs>>
```

```
FROM <tables>, ..., <tname>[size,shift]
WHERE <predicates>, <attr> like <pattern> (p)
```

The main extensions to SQL that we support are: (1) the user has the choice between using MAP or ML operators for converting the final probabilistic answer to a deterministic answer; (2) support for specifying sliding window parameters, and (3) support for pattern queries (including specifying the threshold probability).

## 7.4.2 Query Planning and Optimization

The key challenge in designing a query plan for a given query is avoiding unsafe operators. The two operators that are potentially unsafe (among the operators described in the previous section) are projection and the window aggregate operators. As we discussed above, the sliding window aggregates are always unsafe (since the output itself is of exponential size) and we only compute an approximate answer to those queries (by not computing the correlations in the output sequence). For the tumbling window operator, we separate the final projection step (which may be unsafe) into a separate projection operator. Because of this, the projection operator is the only unsafe operator in our system, and the query planning reduces to determining the correct position for the projection operators in the query plan. Next we present a sketch of our query planning algorithm.

For a given query, we first convert it to a probabilistic sequence algebra expression. We then construct the query plan by instantiating each of the operators with the schemas of their input sequences. Each operator then executes its *schema routine* and computes its output schema, which is used as

the input schema for the next operator in the chain. While doing this, we also check the input to the projection, and determine if the projection operator is safe (see Section 7.3). If a projection-input pair is not safe, we pull up the projection operator through the aggregate and the windowing operators and continue with the rest of the query plan. If the operator we find after the projection is `ML`, then we can determine the exact answer, however if we find a `MAP` operator, we replace both the projection and the `MAP` operator with the approximate-MAP operator (Section 7.4.3) and notify the user that a safe plan cannot be found for the query. After generating a safe query plan as shown here, we optimize it in the next step.

**Example** Suppose that the user issues the query  $Q_0 : \text{SELECT\_MAP MAX}(X) \text{ FROM SEQ WHERE } Y < 20$  on the Markovian sequence shown in Figure 7.1(a). The PSA expression for this query can be written as  $MAP(G^p(\Pi_X^p(\sigma_{Y < 20}^p SEQ)))$ . While running the query planning algorithm on this query, we see that the projection operator immediately after the selection predicate is not safe (illustration is shown in Figure 7.2(b)). Hence, we postpone the projection and execute it after the aggregate operator, to obtain the new plan  $MAP(\Pi_{MAX(A)}^p(G^p(\sigma_{B > 2}^p SEQ)))$ , which is now safe, because the aggregate operator returns a single value.

Our query planning algorithm is both *sound* and *complete*, i.e., we guarantee that the above procedure returns a safe plan if it exists for the query. This is trivial to see because the only reason for not finding safe plans is when the data complexity of the output sequence is *#P-hard* (which happens with unsafe projections and sliding windows).

We optimize the query plan generated above by applying various rules to rearrange operators and to simplify the PGMs generated during query pro-

cessing:

1. *Projection push-down*: If possible (i.e., safe), we push the projections down the query plan. For instance, if the input data streams have no temporal correlations, we can safely execute projections early on.
2. *Exploiting operator commutativity*: If we drop the probabilities (CPDs) in the tuples early, we can reduce the memory cost incurred in storing and routing tuples through the query plan; so we try to push the MAP and the ML operator down the query plan as much as possible. This is in contrast to a traditional database, where we try to push the selection predicate as far down the query plan as possible. Recall that the selection operator commutes with both MAP and ML operators (Section 7.2); hence we can push it down the query plan without affecting the correctness.
3. *Dropping correlations when ML values are requested*: When only the ML values are requested by the user, then we only need to determine marginal distributions for every time instant. Hence, we can drop certain edges in the PGMs of operators that will not influence query results. Suppose that a most likely sequence of the tumbling window aggregate is required by the user. In this case, we can drop edges that exist in the PGM between the first window and the second window because the most likely value sequence is not affected by these edges. For instance, in Figure 7.5(d), if ML values are required, we can drop the dotted edges in the Figure.

### 7.4.3 Approximation Strategies

To execute unsafe operators and to improve the throughput of the aggregate operators, we employ the following two approximation strategies:

#### Approximate MAP operator

We use the Mini-Bucket elimination algorithm of Dechter et al. [33] to approximate the MAP operator. The main idea here is to bound both the dimensionality of the CPDs and the number of CPDs generated during inference. Using this algorithm, we can bound the complexity of a potentially exponential MAP task to be polynomial in the number of tuples. We present results from using this approximation in the Section 7.5.

#### Approximate Aggregates

As described earlier, when the domains of the SUM and COUNT aggregates become large, the throughput of the aggregate operator falls. To counter this, we perform simple approximations beyond a threshold domain size. One such approximation for aggregates/sliding window aggregates is based on simply computing expected values. Using the *linearity of expectation*, we can compute the expected value of SUM and COUNT of aggregates in just  $O(1)$  time. Our system currently does not support approximating AVG aggregate.

## 7.5 Experiments

In this section, we evaluate the efficiency of using our PrDB system for managing and query probabilistic streams. Our experimental results demonstrate that probabilistic stream query processors must incorporate support for temporal correlations, otherwise the query results can be highly inaccurate. We

illustrate the effectiveness of our query processing and optimization algorithms, especially for evaluating aggregate queries over probabilistic streams. We also discuss the trade-offs between accuracy and performance for our approximate MAP operator.

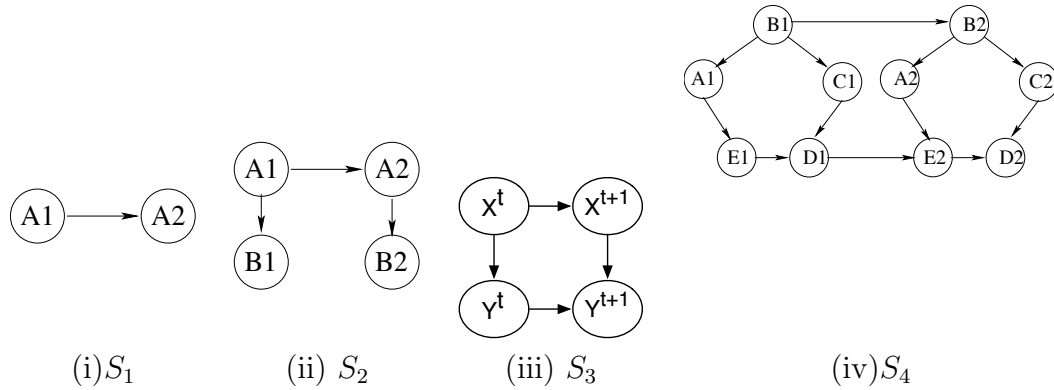
### 7.5.1 Experimental Setup

#### Markovian sequence generator:

We implemented a Markovian sequence generator that generates Markovian sequences for a given input schema. Figure 7.6(a) shows the four schemas that we use in our experiments. Consider the third schema (iii) shown in Figure 7.6(a). For this schema, the generator starts by creating random CPDs for the first time instant -  $p(X^1)$ ,  $p(X^1, Y^1)$ ,  $p(X^1, X^2)$  and  $p(X^1, X^2, Y^2)$ . For each time instant  $t$  after that, it recursively computes  $p(X^t)$  and  $p(X^t, Y^t)$  using the previous CPDs and then randomly generates new CPDs  $p(X^t, X^{t+1})$  and  $p(X^t, X^{t+1}, Y^{t+1})$  for the current time instant. By doing this, we ensure that continuity is maintained for all time instants, i.e., marginals over overlapping random variables in successive joint distributions match up. We also control the amount of spatial and temporal correlations in the sequence using a *correlation coefficient parameter*, which is input to the generator. The domains of the random variables we considered ranged from 3 ( $\{0,1,2\}$ ) to 10 ( $\{0, 1, \dots, 9\}$ ).

We generated such sequences for all schemas shown in Figure 7.6(a). We constructed schema (iv) specifically to denote the sequence generated by our habitat monitoring application with 5 sensor locations and with complex spatial and temporal correlations. We use the notation  $S_i$  to denote the sequence





(a) Different schemas used in the experiments

- Q1: `SELECT_MAP Agg(A) FROM S;`
- Q2: `SELECT_MAP Agg(A) FROM S WHERE B > 1;`
- Q3: `SELECT_MAP MAX(A) FROM S[size,size]`
- Q4: `SELECT_MAP MAX(A) FROM S[size,1]`
- Q5: `SELECT_ML A FROM S2`
- Q6: `SELECT_MAP X FROM S1[size], S3[size]`  
`WHERE S1.A > S3.Y`
- Q7: `SELECT_ML Agg(A) FROM S[10,10]`
- Q8: `SELECT_MAP X FROM S3`

(b) Queries

Figure 7.6: The set of queries and the schemas used in the experiments.

generated from the corresponding schema.

Figure 7.6(b) shows the 8 queries that we use in our experimental evaluation.

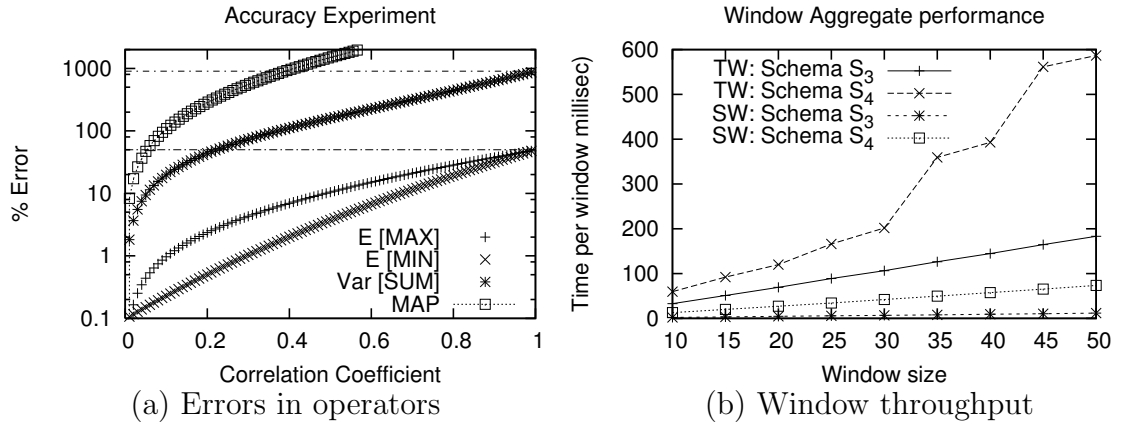
## 7.5.2 Experimental Results

**Query Processing that is aware of temporal correlations is important**

We generate Markovian sequences based on the schema shown in Figure 7.6(ii) for different values of the correlation coefficient parameter  $f$  ranging from 0

to 1. We ensured that the marginal probability of variable  $A$  is  $\{0.5, 0.5\}$  for all time instants. We measure the amount of error for each of our operators when temporal correlations are ignored. For the `MAP` operator, we measure the difference between the probability of the answer returned and the correct probability (the probabilities match only when  $f = 0$ , i.e., when there are no correlations). We plot the error as a function of the correlation coefficient in Figure 7.7(a). We also measured the errors encountered for various aggregates. We plot the error in the expected values of `MIN` and `MAX` and also the errors in the variance of the `SUM` aggregate in the figure (expected value in `SUM` had 0 error as expected). The `PATTERN` operator also suffers lot of error if temporal correlations are ignored (not shown here). As shown in the figure, for correlation coefficients beyond 0.2, the amount of error for all operators is very large if the temporal correlations are ignored.

**Study of Streaming Performance of Aggregates** Here, we measure the throughput of our aggregate and windowing operators. We execute queries  $Q_1$  &  $Q_2$  for all aggregates and measure the time take to process each new tuple completely. From this, we estimate the throughput of our aggregate function. For `MAX` and `MIN` aggregates, the amount of time taken to process new tuples is constant (as expected), however the per tuple processing time increases continuously for `SUM` and `COUNT` as the domains of intermediate results keep increasing. For `SUM` and `COUNT`, we switch to returning expected values (Section 7.4.3) when the domain size exceeds 200. We measure the throughput for both sequences  $S_3$  and  $S_4$ . The results of this experiment are tabulated in Figure 7.7(c). As we can see from the table, our system can support up to 500 tuples/second even with our habitat monitoring schema, which is quite



(a) Errors in operators

(b) Window throughput

Operator	$S_3$		$S_4$	
	$Q_1$	$Q_2$	$Q_1$	$Q_2$
agg_max	2762	1201	509	220
agg_min	2802	1271	559	260
agg_sum	34.5	30.3	15.7	15.6

(c) Aggregate throughput (tuples per second)

Figure 7.7: (a) We plot the % error in query processing for various operators when temporal correlations are ignored, (b) We show performance (throughput) of the windowing operators, (c) We show the throughput of aggregate operators for different cases.

impressive for a system that handles temporal correlations. The value shown for `sum` is the *lowest* throughput we encountered in the experiment. We can improve this number by switching to approximations earlier.

We also measured the throughput of our tumbling window and sliding window operators. We executed queries  $Q_3$  and  $Q_4$  for each of sequences  $S_3$  and  $S_4$  as a function of the size of the window and measured the time taken to process a window of tuples. The results are shown in Figure 7.7(b). As we can see, the processing time increases linearly as a function of the window size. Sliding window processing takes much less time as we perform approximations (Section 7.4.3).

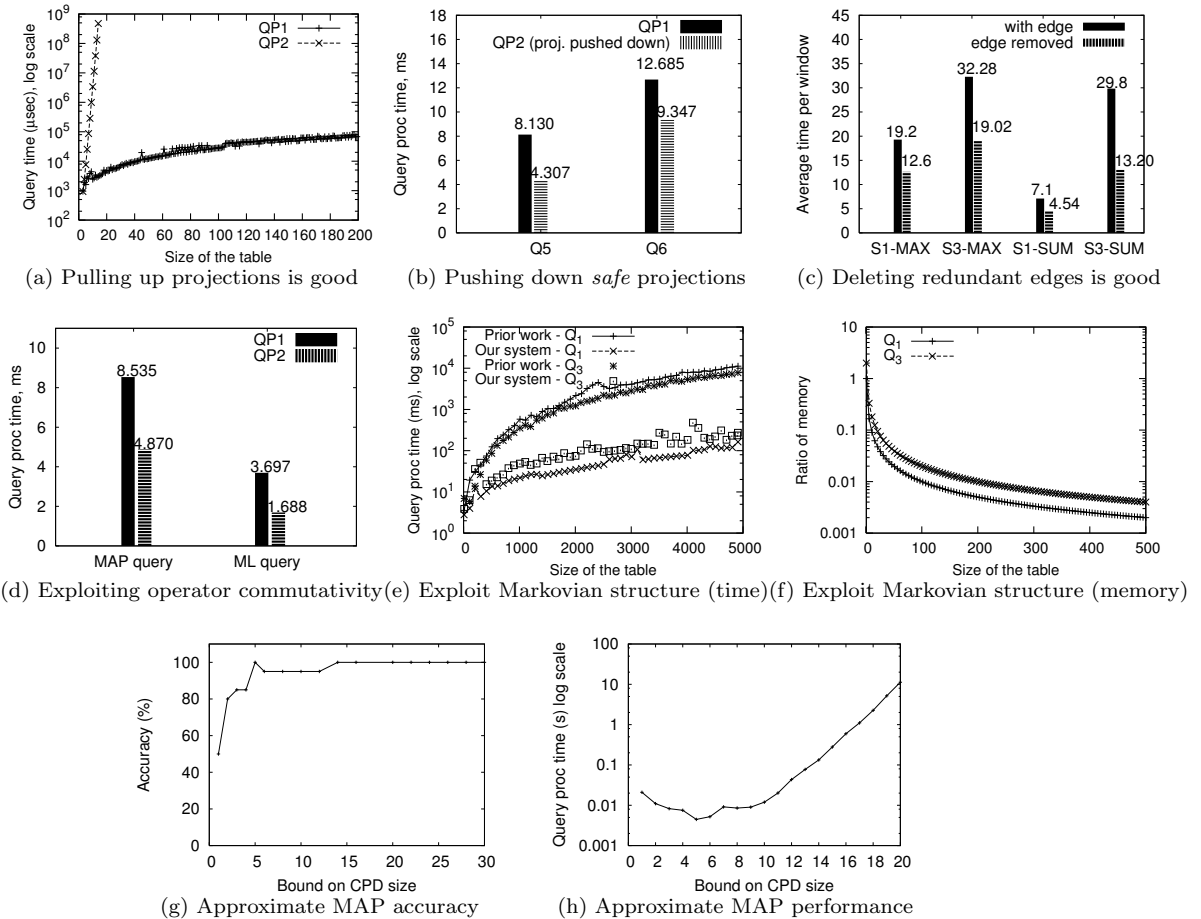


Figure 7.8: Figures (a),(b),(c),(d) illustrate query optimization. (a),(b) show the need for determining the correct location for the projection operator. (c) demonstrates gains made by deleting redundant edges in the model. Note that this is not drawn to scale, only used for comparison. (e),(f) demonstrate advantages of our system over previous approaches. Notice that the y-axis is in the log scale, so our gains are substantial. (g),(h) describe accuracy and performance for the approximate map operator

**Query Optimization Strategies** With this set of experiments, we demonstrate the need for query optimization and effectiveness of our query optimization strategies in choosing efficient query plans.

- **Projections:** Determining the correct position for the projection operator in the query plan is very critical to the query performance. We run query

$Q_0$  shown in Section 7.4.2 with two query plans, QP1, which is obtained by our query optimizer -  $\Pi_{MAX(A)}(Agg_A(TBL))$  (projections pulled up) and the naïve query plan, QP2 -  $Agg_A(\Pi_A(TBL))$  (projections pushed down, as in a standard query optimizer). As Figure 7.8(a) shows, QP2 is extremely inefficient when compared to QP1. In fact, QP2 runs out of memory (1GB) for just 20 tuples. Also note that QP1 is incremental and works well for streams while QP2 requires all the input data at once. Pushing down *safe* projections also helps significantly improve performance (reduced data flow among operators) in many queries. We run queries  $Q_5$  and  $Q_6$  with query plans  $QP_1$  and  $QP_2$  (projections pushed down – for  $Q_6$  we push the projection below join) and compute the total query processing time, which are plotted in the Figure 7.8(b).

- **Dropping edges for ML values:** As we illustrated in Section 7.4.2, we can drop redundant edges in the PGM to improve query performance. We run the ML query  $Q_7$  for SUM, MAX aggregate using two query plans for each query - first one with all edges, the second with edges removed. For analysis, we use both the sequences  $S_1$  and  $S_3$ . The query processing times are plotted in the bar chart in Figure 7.8(c). (We scaled down the query processing times for SUM by 10 to fit the figure). As we can see, we can reduce the query processing time to about half of its value. We observed that even with the habitat monitoring sequence  $S_4$ , we reduced the query processing time by half.
- **Exploiting commutativity of operators:** We examine the amount of savings that we obtain by exploiting the operator commutativity discussed in Section 7.2. We execute query plans  $QP_1$  -  $ML(\sigma^p(PS))$  and  $QP_2$  -

$\sigma(ML(P_S))$  and determine the amount of savings in this case. Similarly, we execute query plans  $MAP(\sigma^p(P_S))$  and  $\sigma(MAP(P_S))$ . First, we verified that the answers returned by both the query plans is indeed the same. Second, we observe about 40% saving in the query processing time if we execute the ML/MAP operator first, as shown in Figure 7.8(d).

**Comparison with previous techniques (Sen et al. [96])** We illustrate that the `get_next()` query processing framework that we have developed is much more efficient than previous approaches. By performing incremental operations, we can not only reduce amount of memory consumed, but also the query processing time. The advantage is magnified while computing aggregates, which create large PGMs. We use queries  $Q_1$  and  $Q_3$  from Figure 7.6 for our experiments. We run the incremental version using our system and then use techniques from Sen et al. [96] to fully construct the PGM for the query. We plot the query processing times, as a function of the table size in Figure 7.8(e). As we can see from the figure, the incremental algorithm is comparable with the previous approaches for small tables, but does better as the size of the table increases. We also have an order of magnitude reduction in the total memory consumed (7.8(f)).

**Approximation Performance and Accuracy** Here, we describe the trade-offs between accuracy and performance provided by the approximate MAP operator (Section 7.4.3). For this experiment, we use Query  $Q_8$  and fix the correlation coefficient to be 0.5. We run the exact version of the query plan first, without any approximations. Since this is exponential, the algorithm could only handle 25 tuples before running out of memory. Then we run the

query plan with the approximate-MAP operator for different bounds on the CPD size. We measure the accuracy of the results by comparing the resulting sequence against the exact MAP sequence computed earlier. The accuracy and the performance benefits of the approx-MAP operator is shown in Figure 7.8(g) and (h). As we can see from the plots, the approximate-MAP operator can be used to obtain fairly accurate query answers.

## Chapter 8

# Robust Query Processing for Probabilistic Databases

So far in the dissertation, we have focused on a simple query evaluation model: given a probabilistic database and a query, we compute the set of output tuples and their probabilities. However, the output provides little or no intuition to the user – for instance, the reason we obtained a given output tuple and why its probability is so low. Providing this information is critical in a probabilistic database since the users may not be sure about the input probabilities and correlation values, and are hence interested to know what tuples are most influential for the output. In this chapter, we augment the existing query evaluation model with two additional features: first *sensitivity analysis* and second *causes/explanations*. Further, we also allow the user to alter input probability values (e.g., if the user resolves uncertainty in some tuple or if the user procures more accurate probabilities) and quickly recompute the query results by exploiting previous computation. The outline of the chapter is as follows. We start with some preliminaries in Section 2.1.7 and formally define



the sensitivity and the explanations problem in Section 8.1. In Section 8.2 and 8.3, we develop algorithms for sensitivity analysis and explanations for various queries. We provide algorithms for incrementally re-evaluating queries in Section 8.4. Finally, we conclude with experimental evaluation in Section 8.5.

## 8.1 Formal Problem Statement

In this section, we formally state the sensitivity analysis and the explanation analysis problems.

### 8.1.1 Sensitivity Analysis

We start by defining the notion of *influence* of a given input probability on a query result. Queries on a probabilistic database can be of two categories based on the type of output: *value* queries and *set* queries. We define influence for each query type in turn.

**Value queries:** The output of a value query is either a single numerical value  $v$  or a set of numerical values  $\{v_1, v_2, \dots, v_n\}$ . Examples of value queries include boolean conjunctive queries (output value is a probability) and aggregation queries (output value is the expected value of the aggregate).

**Definition 3.** *The influence of an input tuple  $t$  on a value query with output  $v$  is given by the derivative  $\frac{\partial v}{\partial p}$ , where  $p$  is the probability that  $t$  exists. If the output is a set of values, then the influence is the sum of the influences of  $t$  on each output value.*

For conjunctive queries, even if we change the probabilities of the input tuples, the output set of tuples remains the same, only the probabilities of

the output tuples change. Thus, the sensitivity of an input probability on a boolean conjunctive query result represents how the output probability changes when the input probability is changed. Influence of an input tuple  $t$  on the output tuple with lineage formula  $\lambda$ , denoted by  $\text{infl}_t(\lambda)$  is given by the derivative  $\frac{\partial p(\lambda)}{\partial p}$ . An alternate definition of influence of a tuple on a conjunctive query, was proposed by Re et al. [92]. Re et al. define influence of a tuple as the difference between the output probabilities obtained in two cases, first by assuming that the tuple exists and second by assuming that the tuple does not exist. As we show in Theorem 2 in Section 8.2.1.1, the two definitions are equivalent.

For aggregation queries, we measure how the *expected* value of the aggregate changes when the input probability is modified, i.e., for the AVG aggregate, we define influence of  $t_i$  to be the derivative  $\frac{\partial \mathbb{E}[\text{AVG}]}{\partial p_i}$ .

**Set Queries:** Examples of Set queries are probabilistic threshold queries and top-k queries. If we change the probability of an input tuple, either new tuples enter the result set or existing tuples leave the result set. To define influence over discrete sets such as these, we introduce the notion of  $\epsilon$ -influence.

**Definition 4.** *Input tuple  $t$  with probability  $p$  is  $\epsilon$ -influential on the output set  $S$  if using  $p + \epsilon$  in place of  $p$  modifies the result from  $S$  to  $S'$ , where  $S' \neq S$ . The degree of influence is the cardinality of the symmetric difference  $S \Delta S' = (S \setminus S') \cup (S' \setminus S)$ .*

Note that  $\epsilon$  is a parameter this is provided by the user. In Section 8.2.3, we show how to provide hints to the user to set  $\epsilon$ .

### **Formal Problem: Sensitivity Analysis**

Given a probabilistic database and a query, determine the set of top- $\ell$  influential/ $\epsilon$ -

influential variables for the query. We use  $\ell$  to distinguish it from conventional top- $k$  queries.

### 8.1.2 Explanation Analysis

Intuitively, an explanation for a query result is a set of tuples which provides the best reason for obtaining the particular result tuple and its associated probability. It is critical to consider a *set* of tuples and their combined contribution rather than contributions by each individual tuple individually. For example, consider an output tuple with lineage  $(a \wedge b) \vee (c \wedge d)$ . Suppose that probability of  $a$  and  $b$  are very high compared to that of  $c$ . The contributions of  $a$  and  $b$  treated individually are very high compared to that of  $c$ . However, the contribution of the set  $\{a, c\}$  is higher than the contribution of  $\{a, b\}$  since we can bring the output probability down to 0 by setting  $a$  and  $c$  to false (which is not possible with  $\{a, b\}$ ). Hence we define the *contribution* for a set of tuples as follows.

**Definition 5.** *A contribution of a set of input tuples  $S$  is defined as the change in the output obtained when we set the probabilities of all tuples in  $S$  to zero.*

For value queries, the change in the output is simply the difference between the two resulting values. For example, in the case of boolean conjunctive queries, this corresponds to the set of tuples which cause the maximum change in the output probabilities if their probabilities are set to 0. For set based queries, the change is the cardinality of the symmetric difference between the resulting sets, as we did with sensitivity analysis. For explanation analysis, we only consider value queries in the rest of the chapter.

#### Formal Problem: Explanation Analysis

Given a probabilistic database and a value query, determine the set of input tuples of size  $\ell$  which has maximum contribution among all subsets of size  $\ell$ .

### 8.1.3 Warmup: SUM/COUNT

As a warmup, we illustrate the concepts we just introduced using SUM/COUNT queries. Using linearity of expectation, we can see that  $\text{SUM} = \mathbb{E}[\sum_i a_i x_i] = \sum_i a_i p_i$ . COUNT is just a special case of SUM where  $a_i = 1 \forall i$ . Therefore, the influence of tuple  $t_i$  is simply  $\frac{\partial \text{SUM}}{\partial p_i} = a_i$ . We can just sort the input tuples by their attribute values (i.e.,  $a_i$ ) and return the top- $\ell$  influential tuples.

It is also easy to see the contribution of the set  $S$  of tuples is  $\sum_{t_i \in S} a_i p_i$ . Therefore, the explanations for COUNT correspond to the  $\ell$  tuples with maximum  $a_i p_i$  values.

Recomputing the results for SUM/COUNT queries is also straightforward. For instance, if we change  $p_i$ , the probability  $t_i$ , to  $p'_i$ , then the new query answer is  $\mathbb{E}[\text{SUM}] - a_i p_i + a_i p'_i$ , which can be done in constant time.

### 8.1.4 Relation to Meliou et al. [77]

Meliou et al. [77] define the notion of responsibility of a tuple  $t$  for a query answer/non-answer as the inverse of the size of the smallest *contingency set* for the tuple. A contingency set is a set of conditions that need to be satisfied for the tuple  $t$  to *cause* a difference to the output. The bigger the contingency set, the smaller is the set of possible worlds that the tuple can influence. If the size of the contingency set is  $s$ , then the total number of possible worlds that can be influenced is of the order of:  $O(\frac{2^n}{2^s - 1})$ ,  $n$  being the number of input tuples. In our definition of influence, we measure the sum of the probabilities

of all possible worlds that are influenced by the given input tuple since all possible worlds need not have the same probability. Hence, for the case of probabilistic databases, the concept of influence is more meaningful, although the two concepts are closely related.

## 8.2 Sensitivity Analysis

In this section, we discuss our algorithms for solving the sensitivity analysis problem, i.e., computing the top- $\ell$  influential variables for a given query. In the first part, we discuss value queries and subsequently we discuss set-based queries.

### 8.2.1 Value queries

We start by developing algorithms to compute influential variables for boolean conjunctive queries and extend the algorithms for arbitrary conjunctive queries. For ease of exposition, we list the following straightforward lemma that is used throughout the chapter. Suppose  $q$  is a value query and suppose that the input tuple probabilities are  $P = \{p_1, p_2, \dots, p_n\}$ . Let  $P_{p_i \leftarrow a}$  be the same vector as  $P$  except the  $i$ th entry being replaced by  $a$ .

**Lemma 1.** *If  $q(P) = q(p_1, \dots, p_n)$  is a linear function of  $p_i$ , i.e.,  $q(P) = c \cdot p_i + d$  where  $c, d$  are constant w.r.t.  $p_i$ , then*

$$\frac{\partial q(P)}{\partial p_i} = \frac{q(P_{p_i \leftarrow a}) - q(P_{p_i \leftarrow b})}{a - b}.$$

*for any  $a, b \in [0, 1]$  and  $a \neq b$ .*

### 8.2.1.1 Boolean conjunctive queries

We first show our definition of the influence of a tuple is equivalent to the definition proposed by Re et al. [92]. Recall that they define influence of a tuple to be the difference between the output probabilities obtained in two cases, first by assuming that the tuple exists and second by assuming that the tuple does not exist.

**Theorem 2.** (1) Given a boolean formula  $\lambda$ , which is a function of input variables  $x_1, x_2, \dots, x_n$ ,  $p(\lambda)$  is linear in each  $p_i$  treated individually, i.e.,  $p(\lambda) = (c_i p_i + c'_i)$  for each  $i$ .

(2) Our definition of the influence is equivalent to the one proposed by Re et al. [92], i.e.,  $\frac{\partial \Pr[\lambda(t)]}{\partial p(t)} = \Pr[\lambda_{x_i=1}] - \Pr[\lambda_{x_i=0}]$ .

*Proof.* Consider a boolean random variable  $x_i$  which appears in the formula  $\lambda$ . Using Shannon expansion,

$$\begin{aligned} \lambda &= (x_i \wedge \lambda_{x_i=1}) \vee (\bar{x}_i \wedge \lambda_{x_i=0}) \\ \implies p(\lambda) &= p(x_i = 1)p(\lambda_{x_i=1}) + p(x_i = 0)p(\lambda_{x_i=0}) \\ &\text{(This is because the two terms are } \textit{mutually exclusive}) \\ \implies p(\lambda) &= p_i p(\lambda_{x_i=1}) + (1 - p_i)p(\lambda_{x_i=0}) \\ \implies p(\lambda) &= p_i c_i + c'_i \end{aligned}$$

Here,  $c_i$  and  $c'_i$  are constants, i.e., independent of  $p_i$ . The second part is a easy consequence of the first part and Lemma 1.  $\square$

As indicated in Section 2.1.7, conjunctive queries are evaluated by first computing the lineages of the output tuples. In the case of boolean queries,

we have a single output lineage for which we need to compute the influential variables. According to Theorem 2, the probability of a boolean formula is linear in each input tuple treated individually, i.e.,  $p(\lambda) = c_i p_i + c'_i$ . Hence, it is enough to determine  $c_i$  values for each input tuple and then select the top- $\ell$  among them. However computing the influence values for all input tuples is #P-complete as shown in Theorem 3.

**Theorem 3.** *The problem of computing the influences of all variables for a non-read-once lineage is #P-complete.*

*Proof.* We prove via a counting reduction from the problem of computing  $p(\lambda)$  where  $\lambda$  is a  $k$ -DNF formula, which is a well known #P-complete problem. Assume that we can indeed compute the influence of all variables on a non-read-once boolean formula  $\lambda$  of size  $n$  in polynomial time. Suppose that the variables in  $\lambda$  are  $x_1, x_2, \dots, x_n$ . Using Theorem 2, can write the probability of  $\lambda$  as:

$$\begin{aligned} p(\lambda) &= p(x_1)p(\lambda_{x_1=1}) + (1 - p(x_1))p(\lambda_{x_1=0}) \\ &= p(x_1)(p(\lambda_{x_1=1}) - p(\lambda_{x_1=0})) + p(\lambda_{x_1=0}) \\ &= p(x_1)\text{infl}_{x_1}(\lambda) + p(\lambda_{x_1=0}) \end{aligned}$$

Note that  $\lambda_{x_1=0}$  is a boolean formula with  $n - 1$  variables.

It can be expanded further.

$$\begin{aligned} &= p(x_1)\text{infl}_{x_1}(\lambda) + p(x_2)\text{infl}_{x_2}(\lambda_{x_1=0}) + p(\lambda_{x_1=x_2=0}) \\ &= p(x_1)\text{infl}_{x_1}(\lambda) + p(x_2)\text{infl}_{x_2}(\lambda_{x_1=0}) + \dots \\ &= \sum_i p(x_i)\text{infl}_{x_i}(\lambda_{x_1=\dots=x_i=0}) \end{aligned}$$

Hence, computing the influences of a variable for an arbitrary DNF is at least as

hard as computing the probability of the formula  $\lambda$  ( $\#P$ -hard). Moreover, since  $\text{infl}_{x_i}(\lambda) = \frac{p(\lambda) - p(\lambda_{x_i=0})}{p(x_i)}$ , our problem is in  $\#P$ . Therefore, it is  $\#P$ -complete.  $\square$

Although the general problem is hard, we can devise algorithms for the special case when the boolean formula is *read-once*. We discuss this case first.

**Read-once lineage:** In this case, we develop a recursive algorithm for computing the influences of all input tuples in  $O(n)$  time ( $n$  is the size of the lineage formula). Consider the lineage shown in Figure 2.8(i). In order to compute the influence of  $x_1$  on the output probability, i.e.,  $\frac{\partial o}{\partial x_1}$ , we can use the chain rule from calculus,

$$\frac{\partial o}{\partial x_1} = \frac{\partial o}{\partial q_1} \frac{\partial q_1}{\partial q_3} \frac{\partial q_3}{\partial x_1} \quad (8.1)$$

The terms on the RHS can be obtained by taking appropriate derivatives of Equations 1 and 2. Suppose  $z$  is a node with two children  $x_1$  and  $x_2$ . Then,

$$\begin{aligned} \text{if } z = x_1 \wedge x_2: & \quad \frac{\partial z}{\partial x_1} = x_2 \ \& \ \frac{\partial z}{\partial x_2} = x_1 \\ \text{if } z = x_1 \vee x_2: & \quad \frac{\partial z}{\partial x_1} = 1 - x_2 \ \& \ \frac{\partial z}{\partial x_2} = 1 - x_1 \end{aligned}$$

---

**Algorithm 3**  $\text{deriv}(x)$ , Read as derivative w.r.t  $x$

---

```

1: if  $\text{parent}(x) = \text{null}$  { $x$  is root} then
2:    $\text{deriv}(x) = 1$ 
3: else
4:   if  $\text{parent}(x)$  is an AND node then
5:      $\text{deriv}(x) = \text{deriv}(\text{parent}(x)) * \text{Pr}(\text{sibling}(x))$ 
6:   else
7:      $\text{deriv}(x) = \text{deriv}(\text{parent}(x)) * (1 - \text{Pr}(\text{sibling}(x)))$ 

```

---

Using the chain rule and the above equations, we develop a recursive algorithm as follows. Each node in the AND/OR tree stores the derivative of the



output probability with respect to itself. We use this to compute the derivatives of its children using the above recursive equations in a top-down manner to finally get the derivative with respect to the leaf nodes (input tuples). Note that the probabilities of all the nodes are precomputed in a single  $O(n)$  pass as a preprocessing step. The relevant snippet of the algorithm is shown in Algorithm 3. After computing the influences of each of the input variables, we determine the top- $\ell$  influential variables either by sorting  $O(n \log n)$  or by making a linear scan over the input tuples  $O(n\ell)$ , based on the value of the input  $\ell$ . We also cache the computed influence values for the input tuples – those can be used for quickly recomputing results in certain cases. Although we illustrated the algorithm for binary trees, our implementation can be easily extended to handle k-ary AND/OR trees.

**Non-read-once lineage:** Next, we consider the sensitivity analysis for non-read-once formulas. We propose a heuristic for evaluating the influences, which is similar to the Dtree construction algorithm of Olteanu et al. [81]. Essentially, we perform a sequence of *Shannon expansions* to expand a non-read-once lineage to a set of *mutually exclusive* read-once formulas. The complexity of the operation is exponential in the *treewidth* [39] of the boolean formula. The complete algorithm is shown in Algorithm 4. We now explain the main aspects of the algorithm.

In Step 1, we check if the boolean formula has a read-once representation using Golumbic’s algorithm (Section 2.1.7.3). If it has a read-once representation, then we use the previous algorithm itself. Otherwise, we expand the boolean formula using Shannon expansion, selecting the variable that appears the most number of times. The expansions of the boolean formula are stored

---

**Algorithm 4**  $\text{infl}(\lambda, \vec{I})$ 

---

**Require:** Boolean formula  $\lambda$ , influence vector  $\vec{I}$

- 1: **if**  $\lambda$  is read-once **then**
  - 2:   **return**  $\text{infl\_read\_once}(\lambda)$
  - 3: **else**
  - 4:   Select boolean variable  $x$  in  $\lambda$  that repeats **most** times
  - 5:   Shannon expansion:  $\lambda = (x \wedge \lambda_{x=1}) \oplus (\bar{x} \wedge \lambda_{x=0})$
  - 6:    $\vec{I} = (1 - p(x)) \text{infl}(\lambda_{x=0}, \vec{I}) + p(x) \text{infl}(\lambda_{x=1}, \vec{I})$
  - 7:    $\vec{I}[x] = \Pr(\lambda_{x=1}) - \Pr(\lambda_{x=0})$  {influence of  $x$  itself}
  - 8: **return**  $\vec{I}$
- 

in a binary tree data structure which we call as a Dtree (after Olteanu et al. [81]). Each node in the Dtree corresponds to a boolean formula. Once we obtain a read-once formula, we stop expanding and compute the influences (local) of all the variables in the formula (Step 2). Over the nodes we execute the Shannon expansion, we also compute the influence for the variable over which using Step 7. Each node in the tree has an *influence vector* of size  $n$ , where  $n$  is the total number of variables in the input lineage. The values in the vector correspond to the *local* influences of the variables on the boolean formula corresponding to the node. This vector is recursively updated, based on the childrens' vectors in Step 6. Finally, the influence vector at the root of the tree has influences of all the variables in the formula.

We illustrate the above algorithm using an example.

**Example 1.** Consider the boolean formula given by  $\lambda = a_1b_1c_1 + a_1b_2c_2 + a_2b_3c_1 + a_3b_4c_1$ . It cannot be represented as a read-once formula. Hence, the algorithm first performs Shannon expansion around  $c_1$  (since it appears 3 times) as shown below.

$$\lambda = c_1(a_1b_1 + a_1b_2c_2 + a_2b_3 + a_3b_4) + \bar{c}_1(a_1b_2c_2) = c_1\lambda_1 + \bar{c}_1\lambda_2$$

We can easily see that both  $\lambda_2 = a_1b_2c_2$  and  $\lambda_1 = a_1(b_1 + b_2c_2) + a_2b_3 + a_3b_4$  are already in read-once format and no more expansion occurs. Now, the influence vectors at  $\lambda_1$  and  $\lambda_2$  are computed based on Algorithm 3. In addition, the influence of  $c_1$ ,  $p(\lambda_1) - p(\lambda_2)$  is computed. Following this, the influence vector at the parent node is updated. Note that since  $a_1$  appears in both  $\lambda_1$  and  $\lambda_2$  nodes of the tree, its influence on  $\lambda$  is available at the influence vector of the root node.

### 8.2.1.2 Conjunctive queries

Here, we consider arbitrary conjunctive queries which return multiple output results. As defined earlier (Section 2.1.7), the influence of an input variable is the sum of its influences on each of the output tuples. Note that even though the output tuples may be in read-once format, the set of output tuples may be correlated with each other, since the input tuples may be shared among the lineages of the output tuples [27]. Hence, we cannot use the naïve approach of looking at the top- $\ell$  set of influential tuples for each output tuple and use them to determine the overall top- $\ell$ . For example, suppose we are interested in determining the top-3 influential tuples for a set of correlated output tuples. A single input tuple might be influential for the output tuples combined, but it may not be enough to appear in the top-3 lists of influential tuples for each of the individual output tuples. However, summing up all the influences would be large enough for it to be in the top-3 list; since the naïve approach does not consider this tuple at all, it fails. Instead, we use a brute force approach where we sum up the influences of a given input tuple on each of the output tuples and pick the top- $\ell$  input tuples based on this value. We are currently working on developing more efficient algorithms for conjunctive queries based

on extensional techniques.

### 8.2.1.3 Aggregation queries

Here, we determine the influence of each input variable on the expected value of the aggregate. We provide algorithms for determining the top- $\ell$  influential variables for MIN/MAX and AVG aggregates.

MIN/MAX: We only consider MAX here. The algorithm for MIN is very similar and we omit it here. We assume all  $a_i$ s are positive and there is a dummy tuple with value 0 and probability 1 to avoid the empty possible world where MAX is undefined. We assume tuples are sorted in a non-increasing order of their scores. Recall  $x_i$  is the indicator variable of the existence of  $t_i$ . It is easy to see that

$$\text{MAX} = \mathbb{E}[\max_i x_i a_i] = \sum_i a_i p_i \prod_{j < i} (1 - p_j).$$

It is easy to see from the above formula that MAX is a linear function of  $p_i$  for any  $i$ . Recall  $P_{p_i \leftarrow a}$  is the same vector as  $P$  except the  $i$ th entry being replaced by  $a$ . By Lemma 1, we have

$$\frac{\partial \text{MAX}(P)}{\partial p_i} = \frac{\text{MAX}(P) - \text{MAX}(P_{p_i \leftarrow 0})}{p_i}.$$

Now, we describe our algorithm. We first show how to compute MAX in linear time. Suppose we denote by  $\max[i, j]$  the maximum of the random tuples  $t_i, t_{i+1}, \dots, t_j$ . If we assume  $a_i \geq a_{i+1} \geq \dots \geq a_j$ , then,  $\max[i, j] = a_i$  with probability  $p_i$  and  $\max[i, j] = \max[i+1, j]$  otherwise. Therefore, we have

that

$$\mathbb{E}[\max[i, j]] = p_i a_i + (1 - p_i) \mathbb{E}[\max[i + 1, j]]. \quad (8.2)$$

When we compute MAX, we store all values  $\mathbb{E}[\max[i, n]]$  for all  $i$ . We can also easily compute  $\prod_{j=1}^i (1 - p_j)$  values for all  $i$  in linear time. Now, we show how to quickly compute  $\text{MAX}(P_{p_i \leftarrow 0})$  for each  $i$  in constant time, provided we have already computed MAX,  $\mathbb{E}[\max[i, n]] \forall i$  and  $\prod_{j=1}^i (1 - p_j) \forall i$ . In fact, it is not hard to see that

$$\begin{aligned} \text{MAX}(P_{p_i \leftarrow 0}) &= \sum_{j < i} a_j p_j \prod_{k < j} (1 - p_k) + \sum_{j > i} a_j p_j \prod_{k < j, k \neq i} (1 - p_k) \\ &= \text{MAX} - \prod_{j \leq i} (1 - p_j) \mathbb{E}[\max[i, n]] \\ &\quad + \prod_{j < i} (1 - p_j) \mathbb{E}[\max[i + 1, n]] \end{aligned}$$

Therefore, the overall running time for finding the top- $\ell$  influential tuples is  $O(n)$ .

**AVG:** Now we consider the problem of computing top- $\ell$  influential variables for AVG. Formally, AVG is defined to be

$$\text{AVG} = \mathbb{E} \left[ \frac{\sum_i a_i x_i}{1 + \sum_i x_i} \right].$$

Note that we have included a dummy tuple with value 0 and probability 1 to keep the denominator non-zero. The following theorem plays a central role for the streaming algorithm in [53] and is also crucial for computing the influence.

**Theorem 4.** ([53]) *For the probabilistic tuples  $t_1, \dots, t_n$ , let  $p_i$  and  $a_i$  be the existence probability and the value of  $t_i$ , respectively. Define function*

$h_{\text{AVG}}(x) = \sum_i a_i p_i x \cdot \prod_{j \neq i} (1 - p_j + p_j x)$ . Then,  $\text{AVG} = \int_0^1 h_{\text{AVG}}(x) dx$ .

From the above theorem, it is easy to see that  $h_{\text{AVG}}(x)$  is linear in  $p_i$ . Since the integral is over  $x$ ,  $\text{AVG}$  is also linear in  $p_i$ . Thus, from Lemma 1, we have

$$\frac{\partial \text{AVG}(P)}{\partial p_i} = \frac{\text{AVG}(P) - \text{AVG}(P_{p_i \leftarrow 0})}{p_i}.$$

It is known that computing  $\text{AVG}$  for a dataset of size  $n$  (in particular, expanding  $h_{\text{AVG}}(x)$ ) can be done in  $O(n \log^2 n)$  time [53]. Computing  $\text{AVG}(P_{p_i \leftarrow 0})$  once  $\text{AVG}$  is already computed additionally takes only linear time given the expansion of  $h_{\text{AVG}}(x)$  (see Section 8.4 on recomputing query results). For each tuple  $t_i$ , we need to compute  $\frac{\partial \text{AVG}}{\partial p_i}$ . Therefore, the overall running time is  $O(n^2)$ .

## 8.2.2 Set queries

In this section, we discuss set-based queries and we develop techniques for computing  $\epsilon$ -influential variables for these queries. We start with probabilistic threshold queries.

### 8.2.2.1 Probabilistic Threshold Queries

To evaluate a probabilistic threshold query  $PT(Q, \tau)$ , we first run the conjunctive query  $Q$  and subsequently select all output tuples with probability more than  $\tau$ . Since the lineage formulas generated by conjunctive queries are monotone and positive [92], increasing the probability of an input tuple can only *increase* the probability of the output tuples, thereby increasing the number of output tuples; similarly, decreasing the probability of an input tuple might remove those output tuples whose probabilities become less than  $\tau$ . We only consider the case when the input probabilities are increased. The

symmetric case of decreasing the input probabilities is analogous and is not discussed here. Our objective is to compute the top- $\ell$   $\epsilon$ -influential variables (Section 2.1.7). We rank an input tuple by its degree of  $\epsilon$ -influence, i.e., the number of output tuples that will be added to the output set, if we increase its probability by  $\epsilon$  (Definition 4).

We briefly discuss the naive algorithm before presenting techniques for improving it. In the first step, we compute the output tuple lineages and subsequently, the influences of each input tuple on each output tuple. In the second step, we go over each input tuple ( $t_i$ ) and determine the number of output tuples ( $C_i$ ) that would cross the threshold if we increase its probability by  $\epsilon$ . Finally, we compute the top- $\ell$  influential input tuples from this list. This technique is quite inefficient as the complexity of the first step is  $O(no)$ , where  $o$  is the number of output tuples and  $n$  can potentially include all input tuples. We reduce the complexity of this operation by considering only those output tuples which contribute to the  $C_i$  values, by developing three pruning rules.

**Rule 1:** Ignore output tuples with probability  $> \tau$  – increasing the probability of its input tuples will not change the output.

**Rule 2:** Restrict attention to output tuples with probability  $> \tau - \epsilon$ . The influence of an input tuple over a single output tuple for conjunctive queries is always less than 1, since influence is defined as a difference between two probability values. Therefore, the probability of an output tuple can at most increase by  $\epsilon$  and output tuples with probability values less than  $\tau - \epsilon$  cannot get into the result set.

Next, we propose another pruning rule which works only for read-once lineages. Consider output tuple  $O_i$ . Suppose it has probability  $o_i < \tau$ . The input

tuples that can drive this probability to  $\tau$  must have influence at least equal to  $\theta = (\tau - o_i)/\epsilon$ . So, we only need to increment  $C_j$  values of those input tuples whose influences exceed  $\theta$ . We modify the *infl\_read\_once* routine of Section 8.2.1.1 to only return the input tuples that satisfy the above requirement by exploiting the following property:

**Theorem 5.** *The derivatives of the nodes in an AND/OR tree monotonically decrease as we go down the AND/OR tree.*

*Proof.* The proof is very easy to see using recursion. Recall the recursive equations used to update the probabilities of internal nodes in the AND/OR tree.

$$\begin{aligned} \text{if } z = x_1 \wedge x_2 & \quad \frac{\partial z}{\partial x_1} = x_2 < 1 \\ \text{if } z = x_1 \vee x_2 & \quad \frac{\partial z}{\partial x_1} = (1 - x_2) < 1 \end{aligned}$$

Now, using Equation (3), we can see that the values of the derivatives decrease as we go down the tree since we continuously multiply by a number less than 1 at each level. The derivative of the root with respect to itself is 1 by definition.

□

**Rule 3:** In *infl\_read\_once*, if the value of the derivative is less than  $\theta$ , we do not recurse along the branch. We only recurse along the portion of the tree whose derivative is more than  $\theta$ . Since the derivatives are computed in a top-down manner, pruning via this rule can provide several benefits for large data sets.



### 8.2.2.2 Top-k queries by probability

The output of a top-k query is a list of output tuples sorted in decreasing order by their probabilities. Therefore, modifying input tuples can cause new tuples to enter the output while simultaneously removing the same number of existing output tuples. Suppose we treat the probability of the  $k^{\text{th}}$  output tuple (in order) as the threshold  $\tau$ . As with probabilistic threshold queries, the only tuples that can enter the result are the set of output tuples with probabilities in the range  $[\tau - \epsilon, \tau]$ . Hence, we can apply the pruning rules 1 and 2 work here also. Hence the only input tuples which are influential are the ones that appear in the lineages of these tuples. The algorithm for computing influential variables here is similar to the one for probabilistic threshold queries. The difference between top-k and probabilistic threshold queries is that, for an input tuple  $t$  to force the output tuple  $O_i$  into the top-k output, it is not enough that its influence value exceed  $(\tau - o_i)/\epsilon$ . The reasons are two fold. First, by increasing the probability of an input tuple, we may be increasing the threshold  $\tau$  itself, i.e., if the input tuple appears in the lineage of the  $k^{\text{th}}$  ranked tuple. Second, once a new tuple enters the top-k, the value of  $\tau$  needs to be increased. Hence, we have to explicitly check the number of new tuples entering the top-k for each input tuple and then compute the top- $\ell$  influential input tuples.

### 8.2.3 How is $\epsilon$ assigned?

Until now, we assumed that  $\epsilon$  was provided by the user. However, it is unlikely to expect the user to know the value of  $\epsilon$ . Firstly, a user might know the margin of error that might be present in the input probabilities. For

instance, if the application generating the input probability reports that there may be  $\pm\delta$  error in the probabilities, then the user can pick  $\epsilon$  between  $[0, \delta]$ . Another way to pick a reasonable  $\epsilon$  value might be through a visualization tool. Given an input tuple  $t$  with probability  $p$ , the output probabilities are linear in  $p$  and can be visualized as straight lines. The user can now pick  $\epsilon$  using this visualization, e.g., a region with several intersections.

## 8.3 Explanation Analysis

In this section, we provide algorithms for computing the top- $\ell$  explanations for value queries.

### 8.3.1 Boolean Conjunctive Queries

Recall that computing explanations requires us to determine the set of  $\ell$  input tuples, whose probabilities when set to 0, causes the maximum decrease in the output probabilities.

**Theorem 6.** *The problem of computing the top- $\ell$  explanations for boolean conjunctive queries (even without self-joins) is NP-hard.*

*Proof.* We use a reduction from vertex cover on 3-uniform 3-partite hypergraph [99] similar to Theorem 4.1 of Meliou et al. [77]. Consider a 3-uniform 3-partite graph  $G$  with partitions  $R$ ,  $S$  and  $T$ . Every hyperedge contains exactly one vertex from each partition. We construct a database  $D$  with 4 relations  $R(x)$ ,  $S(y)$ ,  $T(z)$  and  $U(x, y, z)$ . For each vertex in  $R$ ,  $S$  and  $T$ , we introduce a tuple (with tuple-uncertainty) in  $R(x)$ ,  $S(y)$  and  $T(z)$  respectively. For each hyperedge in  $G$ , we introduce a new deterministic tuple in  $U(x, y, z)$ .

Consider the boolean conjunctive query:

$$q() : -R(x), S(y), T(z), U(x, y, z)$$

We can easily see there is a vertex cover of size at most  $\ell$  in  $G$ , if and only if we can determine an explanation of size  $\ell$  that reduces the probability of the output tuple to zero (maximum possible reduction) in the probability.

□

Although the problem is hard in general, for queries that lead to read-once lineage formulas, there exists an optimal algorithm. We describe this algorithm next.

### **Read-once formulas**

For the case of read-once functions, we use a dynamic programming algorithm to compute the explanation. Consider a lineage formula  $\lambda$  with two subtrees  $\lambda_l$  and  $\lambda_r$ . Suppose the function  $OPT(\lambda, k)$  represents the smallest possible value for probability of  $\lambda$  by setting  $k$  input probabilities to 0 (i.e., maximum possible reduction in the probability). The appropriate recursion for the dynamic program is shown below.

$$\underline{\text{If } \lambda = \lambda_l \wedge \lambda_r,}$$

$$OPT(\lambda, k) = \min_{k_1} (OPT(\lambda_l, k_1) \times OPT(\lambda_r, k - k_1))$$

$$\underline{\text{If } \lambda = \lambda_l \vee \lambda_r,}$$

$$OPT(\lambda, k) = \min_{k_1} \left[ \begin{array}{l} OPT(\lambda_l, k_1) + OPT(\lambda_r, k - k_1) \\ -OPT(\lambda_l, k_1) \times OPT(\lambda_r, k - k_1) \end{array} \right]$$

We modify the above program to also include the top- $\ell$  input tuples at each step. The proof of correctness of the above program can be seen via contradiction. Assume, for the sake of contradiction that there exists a different solution  $S'$  (set of  $\ell$  variables) which is better than the solution obtained by the algorithm  $S$ . We consider two cases. Suppose the root node is an AND node with two children  $\lambda_l$  and  $\lambda_r$ . Also suppose  $S'$  has  $l_1$  variables on the left child and  $l - l_1$  variables on the right child, which are different from  $S$ . Denote  $\Pr(\lambda, S)$  the probability of the boolean formula  $\lambda$  by setting the probabilities of variables in  $S$  to 0. According to our assumption  $\Pr(\lambda, S') < \Pr(\lambda, S)$ . This means that  $\Pr(\lambda_l, S'_l)\Pr(\lambda_r, S'_r) < \Pr(\lambda_l, S_l)\Pr(\lambda_r, S_r)$ . However, this statement is false since the sets  $S_l$  and  $S_r$  were chosen such that their product was minimum (according to our update rule).

The complexity of the above program is  $O(n\ell^2)$  since at each step we spend time  $O(\ell)$  to determine  $OPT(\lambda, \ell)$  and we need to compute  $OPT$  for different values from  $\{1, 2, \dots, \ell\}$ .

### Non-read-once formulas

We propose two greedy heuristics for this problem.

**First Approach:** For each input tuple  $t_i$ , we compute its influence  $\alpha_i$  for the output tuple and sort the tuples by the value of  $\alpha_i p_i$ . We select the top- $\ell$  tuples as the best explanation for our purpose. The motivation for this heuristic is that the product of the influence and the probability corresponds to the “individual contribution” made by the tuple (to a first approximation). Note that this ignores the pairwise and higher order contributions.

**Second Approach:** Now, we propose a slightly better approximation. In the first step, we select the tuple with the highest contribution, i.e.,  $\alpha_i p_i$  value.

Next, we set this tuple probability to zero and recompute the contributions for the remaining tuples. We pick the tuple with the highest contribution and repeat the process  $\ell$  times.

### 8.3.2 Aggregation queries

We describe how to compute explanations for MAX and MIN efficiently. For AVG, we leave it as an open problem.

MAX Again, we assume all values are positive and the maximum value is 0 if no tuple exists. We first note here that greedy algorithms do not work in this case, i.e., selecting the tuples sorted by score  $a_i$  or sorted by probability  $p_i$  or sorted by  $a_i p_i$  values. For example, consider the set of four tuples  $X_1 = 10$ ,  $X_2 = 9$ ,  $X_3 = 5.1$  and  $X_4 = 3$  with probabilities 0.1, 0.9, 0.2 and 0.3 respectively. Choosing the greedy heuristic based on the  $a_i p_i$  value would force us to choose  $X_2$  and  $X_3$ , however it can be seen that the optimal solution here is by choosing  $X_1$  and  $X_2$ .

We propose a dynamic program for this problem. Firstly, note that in the MAX case, the expected value can only reduce if we set probability values to zero. Also, if we set more tuple probabilities to zero, the expectation can only come down. Hence, the optimal solution will have exactly  $\ell$  tuples. The following recursive relationship is particular useful to us. Suppose we denote  $\max[i, j]$  as the maximum of the random tuples  $t_i, t_{i+1}, \dots, t_j$ . We also assume  $a_1 \geq a_2 \geq \dots, a_n$ .

Now, we describe our dynamic program. Let  $OPT(i, j)$  denotes the minimum expectation for the maximum of  $\{t_i, \dots, t_n\}$  by reducing the probabilities of  $j$  tuples to zero. Suppose we had optimal solutions to the sub-problem

$\{t_{i+1}, t_{i+2}, \dots, t_n\}$  for all different values of  $k$ , i.e.,  $OPT(i+1, j)$  for  $j = 1 \dots \ell$ . Then, we update the optimal solution using the following recursive equation.

$$OPT(i, j) = \min\{OPT(i+1, j), (1 - p_i)OPT(i+1, j-1) + p_i a_i\}$$

The first term in the right hand side of the recursion corresponds to that  $t_i$  is not chosen while the second corresponds otherwise. The second holds because of (8.2). The optimal solution is simply  $OPT(1, n)$ .

The dynamic program maintains an array of size  $n \times \ell$ , which maintains the optimal solution  $OPT(i, j)$  for each value of  $i$  starting from  $n$  to 1. Computing each entry needs constant time. Therefore, the overall running time is  $O(n\ell + n \log n)$ .

**MIN:** We make the same assumption as **MAX** that all values are positive and the minimum value is 0 if no tuple exists. The dynamic programming recursion is very similar to the case for **MAX**. The only difference is that the expected value can either increase or decrease by setting certain probabilities to 0. Therefore, we use two tables, one for computing the maximum increase and the other for the maximum decrease. Then, we pick the one with larger absolute value as the final answer.

## 8.4 Incremental Recomputation

In this section, we describe how our techniques for computing influences can be used to recompute the results of a query when some of the input probabilities are modified. Note that in addition to query results, we need to update the influences of the input tuples also. It is desirable that the

cost for recomputing query results be less than executing the query again from scratch. We propose efficient *incremental* algorithms for recomputing query results which exploit previous computation. We start by describing the algorithm for updating conjunctive query results.

### 8.4.1 Conjunctive Queries

We consider boolean conjunctive queries. Handling set-based conjunctive queries is a very simple extension and we do not explain this case, owing to space constraints. To recompute answers to conjunctive queries, we use the values of the gradient that we computed while determining the influential variables. For instance, if the user changes the value of a *single* input tuple  $X_i$  from  $p_i$  to  $p'_i$ , then we can use the previously computed derivative to compute the new answer probability in  $O(1)$  time :

$$p^{new} = p^{old} + \frac{\partial p}{\partial p_i}(p'_i - p_i)$$

Obviously, this works only when exactly one input tuple probability is modified. When multiple input tuple probabilities are modified, we can efficiently update the results for read-once lineages.

**Read-once Lineages:** Suppose that the user modifies  $c$  input tuple probabilities. Then, we can update the output probabilities in time  $O(c \log(\ell))$ , where  $\ell$  is the size of the lineage. We illustrate this algorithm below. As we mentioned before, we store the AND/OR tree which was initially generated for executing the query. We first construct a *Steiner tree* in the AND/OR tree connecting the input tuples that are modified by the user and the root of the tree. We subsequently update the probabilities of each of the nodes contained

in the AND/OR tree using a bottom-up algorithm. Each node sends its old probability and new probability to its parent, based on which the parent determines its new probability and sends its old and new probability values to its parents recursively. The update procedure is given below: Suppose that  $p$  is the parent of node  $x$ , which sends  $x^{old}$  and  $x^{new}$  to it. Then node  $p$  executes the following routine.

---

**Algorithm 5**  $\text{update}(x^{new}, x^{old})$

---

- 1: **if**  $p$  is an AND node **then**
  - 2:    $p^{new} = p^{old} \frac{x^{new}}{x^{old}}$
  - 3: **else**
  - 4:    $p^{new} = 1 - (1 - p^{old}) \frac{1-x^{new}}{1-x^{old}}$
  - 5: Send  $p^{old}$  and  $p^{new}$  to parent of  $p$
- 

We illustrate the algorithm with a simple example. Suppose that the user updates the probabilities of the input tuples  $x_1$  and  $x_4$  in Figure 2.8. In that case, a Steiner tree is constructed, connecting  $x_1$ ,  $x_4$  and  $o$ . After this, the probabilities of nodes  $q_3$  and  $q_4$  are updated using the equations described above. Following this, the probabilities of the nodes  $q_1$  and  $o$  is updated. The complexity of the above operation is  $O(ch)$  where  $c$  is the number of nodes that are updated and  $h$  is the height of the tree. If a substantial number  $O(n)$  of input probabilities are updated, we instead use the linear algorithm of Section 2.1.7.1.

Once we modify the probability of a tuple, the derivatives corresponding to each of the other nodes change and we also need to update them. We propose a lazy technique for updating the probabilities. For simplicity, suppose that only one variable is updated. As described earlier, after computing the path from the modified node towards the root, we update their probabilities. However,



note that the derivatives for each of these nodes remain the same. We need to update the derivatives for the other nodes in the tree, which is at least linear in the size of the tree. Instead, we simply mark those nodes, whose children's derivatives are inconsistent. To actually update the derivatives, we adopt a recursive top-down strategy where we update the derivative of the node based on the probability of the parent. If several tuples are modified, we batch together multiple updates and perform the derivative update simultaneously in  $O(n)$  time.

**Non-read-once Lineage:** To update the probability of a non-read-once lineage, we exploit the binary tree data structure that we generated while compute the influences (See Section 8.2.1.1). For simplicity, suppose that the user modifies the input probability of a tuple  $t$ . Since the variable  $x$  corresponding to  $t$  might appear in several portions of the tree, we need to essentially update all portions of the tree that contain  $x$ . Hence, we recurse over the binary tree top-down over the nodes that contain  $x$ . Note that we can use the influence vector in order to determine whether a node contains  $x$  by simply checking if its influence value is 0. Once the children update the probabilities, we update the probabilities of the parent. We also update the influences of the variables. We exclude the details of updating the influences owing to space constraints.

## 8.4.2 Aggregation

Now, we discuss the problem of incrementally re-evaluating the results of aggregation queries, specifically MIN/MAX and AVG.

**MAX:** For MAX queries, our result is a dynamic data structure DS such that

1. The MAX query can be answered from DS in constant time,

2. We need  $O(n)$  time to build DS from scratch.
3. If the probability of a tuple gets changed, we need  $O(\log n)$  time to update DS.

Recall the notation  $\max[i, j]$  denotes the maximum of the random tuples  $t_i, t_{i+1}, \dots, t_j$ . We assume  $a_1 \geq a_2 \geq \dots \geq a_n$  where  $a_i$  is the score of tuple  $t_i$ . Let  $P[i, k] = \prod_{x=i}^k (1 - p_x)$ . We can easily show the following generalization of (8.2) by induction (proof omitted here): For any  $i \leq k \leq j$ ,

$$\mathbb{E}[\max[i, j]] = \mathbb{E}[\max[i, k]] + P[i, k]\mathbb{E}[\max[k + 1, j]]. \quad (8.3)$$

DS makes use of interval trees (see e.g., [29]) which we briefly describe as follows. An interval tree  $\mathcal{T}$  is a binary tree where each node represents an interval  $[i, j]$  for some integer  $i \leq j$ . The root corresponds  $[1, n]$ . For a node  $[i, j]$ , its left child and right child represent  $[i, \lfloor \frac{i+j}{2} \rfloor]$ ,  $[\lfloor \frac{i+j}{2} \rfloor + 1, j]$ , respectively. The leaves of  $\mathcal{T}$  correspond to singletons. It is easy to see that such a tree with  $n$  nodes has height  $O(\log n)$ .

DS consists of two interval trees  $\mathcal{T}_P$  and  $\mathcal{T}_E$ , the first used for maintaining the information of  $\mathbb{E}[\max[i, j]]$ s and the second for  $P[i, j]$ s. In other words, node  $[i, j]$  in  $\mathcal{T}_P$  ( $\mathcal{T}_E$ ) stores the value of  $P[i, j]$  ( $\mathbb{E}[\max[i, j]]$ ). Assuming we have constructed  $\mathcal{T}_P$  and  $\mathcal{T}_E$ , the answer to the MAX query is just  $\mathbb{E}[\max[1, n]]$  which can be retrieved in constant time. It is also not hard to show that both  $\mathcal{T}_P$  and  $\mathcal{T}_E$  can be constructed in linear time. We just start from leaves and build the trees bottom up using formulas  $P[i, j] = P[i, k]P[k + 1, j], i \leq k \leq j$  and (8.3). Now, we describe how to do updating operation in  $O(\log n)$  time for  $\mathcal{T}_P$ . Suppose we update the probability of a leaf  $v$  (which corresponds to a singleton tuple). The new  $P$  value for that node is trivial to compute. The

key observation is only the nodes on the path from  $v$  to the root need updates and the  $P$  values for any other nodes remain the same because their intervals do not intersect with that of  $v$ . The updates can be done bottom up from  $v$  to the root and take at most  $O(\log n)$  times. The updating operation for  $\mathcal{T}_E$  is the same as for  $\mathcal{T}_P$ , except that in each update we need some value  $P[i, j]$  (recall we use (8.3) to update the values). But fortunately, such a  $P[i, j]$  can be readily retrieved from the corresponding node in  $\mathcal{T}_P$  in constant time (for this purpose, we need for each node  $[i, j]$  in  $\mathcal{T}_E$  a pointer to node  $[i, j]$  in  $\mathcal{T}_P$ ). The procedure for MIN is similar to that of MAX and is omitted.

**AVG:** Now, we discuss how to recompute the query result for AVG query. Our algorithm needs an  $O(n \log^2 n)$  preprocessing time and  $O(n)$  time for each probability update. Recall function  $h_{\text{AVG}}(x) = \sum_i a_i \cdot \prod_{j \neq i} (1 - p_j + p_j x)$  and  $\text{AVG} = \int_0^1 h_{\text{AVG}}(x) dx$  (see Theorem 4). The algorithm maintain the expansions of the two polynomials  $h_{\text{AVG}}(x)$  and  $P(x) = \prod_j (1 - p_j + p_j x)$ . Initially, the expansion of  $h_{\text{AVG}}(x)$  can be computed in  $O(n \log^2 n)$  time using the algorithm from [53]. The expansion of  $\prod_j (1 - p_j + p_j x)$  can be computed similarly using the same time. For each update of  $p_i$ , we recompute  $P(x)$  as follows: Suppose the the old and new probabilities of  $t_i$  are  $p_i$  and  $p'_i$ , respectively.

$$P(x) \leftarrow P(x) \frac{1 - p'_i + p'_i x}{1 - p_i + p_i x}.$$

$h_{\text{AVG}}(x)$  can be recomputed as follows:

$$\begin{aligned} h_{\text{AVG}}(x) \leftarrow & \left( h_{\text{AVG}}(x) - a_i p_i x \prod_{j \neq i} (1 - p_j + p_j x) \right) \frac{1 - p'_i + p'_i x}{1 - p_i + p_i x} \\ & + a_i p_i x \prod_{j \neq i} (1 - p_j + p_j x) \end{aligned}$$

where  $\prod_{j \neq i} (1 - p_j + p_j x)$  can be computed from  $P(x)$  in linear time. We can easily see other operations also run in linear times. Thus the overall updating time is  $O(n)$ .

## 8.5 Experimental Evaluation

The main objectives of our experimental analysis are to show:(1) sensitivity analysis is critical for probabilistic databases, (2) sensitivity analysis can be performed at low overhead, (3) explanations can be performed efficiently and (4) incremental recomputation of query results is efficient. We focus on conjunctive queries to illustrate the above points. We implement our system using JDK 1.6. We use MySQL to store the relations in our database. All experiments were run on a 2.4Ghz Core 2 Duo machine with 2GB of main memory. We begin with a discussion of the experimental setup.

**Dataset:** We synthesized a 100 MB TPC-H dataset augmented with tuple uncertainty for each tuple (lineage is stored as a separate column). The probabilities of existence were chosen uniformly between  $[0, 1]$ . To speed up computation, we build indexes on the primary and foreign key attributes of each of the relations.

**Queries:** For experiments on conjunctive queries and probabilistic threshold queries, we used TPC-H queries Q2,Q3,Q5,Q7,Q8 and Q10. We omitted the queries over one relation because of their simplicity. For each of these queries, we removed all aggregation constructs. In addition, we generated boolean versions of these queries by projecting the final outputs to 1, the resulting queries are respectively labeled R2, R3, R5, R7, R8 and R10. Lineages for the output tuples are computed using a query rewrite-based approach shown in

### 8.5.1 Experimental Results

**Sensitivity analysis is essential and critical:** Queries over probabilistic databases are highly sensitive to input tuple probabilities. The sensitivity is even more pronounced for queries that return sets. We use a top-k query by probability to illustrate this point. We first execute the corresponding conjunctive query and compute the influences of all the input tuples on the output tuples. Then we extract a fragment of the set of output tuples and plot their probabilities against a particular input tuple probability  $x$  in Figure 8.1(a). As shown earlier, all output probabilities are linear functions of  $x$ . Tuple  $O4$  did not contain  $x$ . Therefore, its probability was constant. As we vary the probability of  $x$ , notice that the top-k list changes significantly. If the  $p(x)$  is near 0.6, then the top-k list changes if we increase or decrease its probability by a small amount. Hence there is a need for sensitivity analysis to verify query results and provide robust query processing capability.

**Overhead of sensitivity analysis is small:** Now, we want to show that sensitivity analysis can be performed efficiently. When a user marks a query for sensitivity analysis, we not only have to compute query results, but also the influential variables. We measure the *overhead* of computing influential variables over computing just the query result probabilities. We measure this overhead for the set of TPC-H queries mentioned before, and their boolean versions. The results are shown in Figures 8.1(b), (c) and (d). As shown in Figure 8.1(b), the overhead involved in computing influential variables is very small, less than 5% in all queries considered. Note that this is true even for *un-*

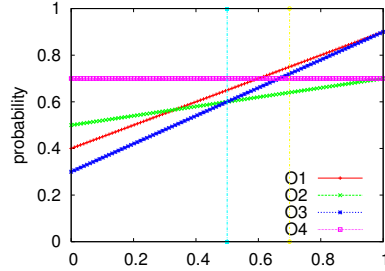
*safe* queries Q7 and Q8. We now study the time taken for different components of the sensitivity analysis. The breakdown of the times for different components is shown in Figure 8.1(c). As shown in the figure, one of the significant time consuming steps is the computation of the lineage itself (indicated by red bars with large crossings) and the time for building the Dtree (Section 8.2.1.1, indicated by green bars with tiny squares). The time taken for computing output probabilities and for sensitivity analysis (S.A.) are mostly comparable. Note that for read-once lineages (Q2,Q3,Q8,Q10), the time taken for lineage computation is the most dominating factor. For queries generating non-read-once lineages, the time taken to compute the appropriate Dtree is the most dominating factor (Exponential complexity). Once the Dtree is constructed, computing the probability and the influential variables is fairly quick. We observe similar results for boolean conjunctive queries in Figure 8.1(d). Except for R2, every boolean query generated a non-read-once lineage. The overheads are slightly higher for boolean queries since we only need to compute a single probability (unlike conjunctive queries), but multiple influence values (one for each input tuples).

**Pruning:** In this experiment, we study the performance of our pruning rules. We used the probabilistic threshold query:TPC-H query Q2 with threshold 0.7 and selected different values of epsilon, from 0.1 to 0.7. We evaluate the naive query+influence times versus the query+influence times obtained by using all the three pruning rules of Section 8.2.2. The results are shown in Figure 8.1(e). As shown in the figure, for small values of  $\epsilon$ , our pruning rules bring down the evaluation time by about 50%. When the value of  $\epsilon$  increases beyond a point, we need to look at every output tuple, hence the performance drops, ultimately

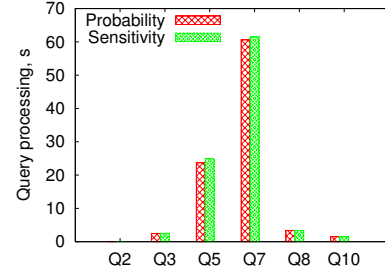
to that of the naive strategy.

**Computing explanations is efficient:** In this experiment, we study the performance of our algorithms that compute explanations. We vary the size of the lineage and measure the time taken to compute the explanations. We experiment with different sizes of explanations from 2 to 10. Our results are shown in Figure 8.1(f). We note here that we only considered read-once lineages for the experiment since the greedy heuristics that were proposed (Section 8.3) for non-read-once lineages are very efficient. According to Section 8.3, computing explanations is linear in the size of the lineage. This was experimentally verified as shown in Figure 8.1(f) (we plot the figure after fitting a line over the data points, actual points are not shown). As we can see from the figure, even for fairly large lineage formulas, computing explanations is quite fast and is comparable to the actual query execution times.

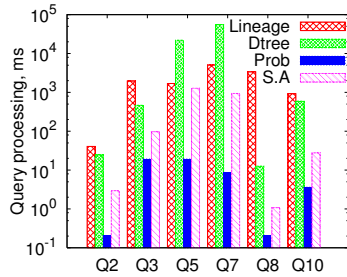
**Study of incremental recomputation of output probabilities:** In this experiment, we study the performance of incremental re-evaluation of output tuple probabilities when input probabilities are modified. For lineage formulas of different sizes, we modify the input tuple probabilities and compute the time (a) for completely re-evaluating the probability from scratch, and (b) incrementally recomputing the probabilities as described in Section 8.4. We evaluate three cases in which we modify 10, 20 and 50 input tuple probabilities. The results are shown in Figure 8.1(h). As shown in the figure, the time taken for incremental recomputation is an order of magnitude lesser, even when we modify upto 50 input tuple probabilities (Please note that the y-axis is a log plot). This illustrates the advantages of our incremental re-evaluation approach.



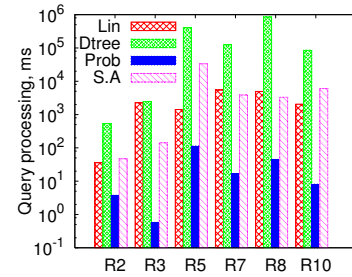
(a) top-k queries (by prob) are very sensitive



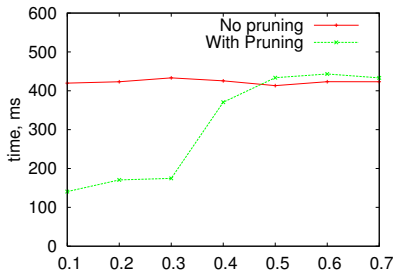
(b) Overheads for TPC-H queries



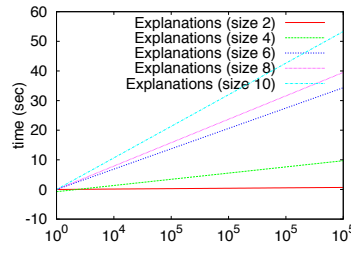
(c) Query proc time break-up (TPC-H)



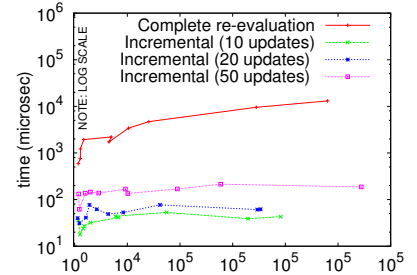
(d) Query proc time break-up (boolean)



(e) Pruning helps when  $\epsilon$  is small



(f) Explanations vs lineage size



(g) Incremental recomputation

Figure 8.1: Results: (a) Top-k queries by probability are sensitive to input probabilities. As we modify the probability  $p(x)$ , the output probabilities and the top-k output change as shown. (b,c) Here we demonstrate that sensitivity analysis can be implemented very efficiently. The overhead above computing output probabilities for TPC-H queries is at most 5%. (d) The break up of the times spent in the different components of the algorithm. S.A refers to sensitivity analysis. (e) Same as part(d) for Boolean TPC-H queries (f) Illustration of the benefits of pruning algorithms. (g) Explanation analysis is efficient. (h) Incremental recomputation for boolean conjunctive queries is efficient.



# Chapter 9

## Conclusions

Advances in miniaturization technology and sensing 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 addition, web-based applications such as information extraction, data integration, sentiment analysis and other machine learning applications generate increasing amounts of data. Much of it however, is uncertain and incomplete due to a number of reasons including inaccuracies in measurements, sensor node failures and so on. Developing scalable query processing techniques over such data has become an important task in database research. In this dissertation we identified the challenges involved in managing large-scale correlated probabilistic data and developed a few tools and techniques for building a database system for managing such data. In this chapter, we briefly outline the main contributions made by the dissertation.

- We started with a description of probabilistic modeling, which is a (necessary) pre-processing step for querying uncertain data. We presented an approach to build an extensible database system for enabling 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 developed intuitive interfaces to declaratively specify the models to be applied. The output of such probabilistic modeling is a probabilistic database which we store using a probabilistic graphical model.

- Next, we developed a representation for correlated probabilistic databases using junction trees and adapted the message passing algorithms (*belief propagation*) for evaluating queries directly over junction trees. We developed algorithms for *inference/what-if* queries, *aggregation* queries and *conjunctive queries*. While the general problem of conjunctive query evaluation is  $\#P$ -complete, we developed heuristics that scale to large junction trees.
- Next, we developed an index data structure (INDSEP) for correlated probabilistic databases which allows for efficient query evaluation. The key component of INDSEP is the *shortcut potential*, which allows us to speed up belief propagation by shortcutting across large sections of the junction tree. Using INDSEP we not only scaled up query evaluation algorithms to work for very large-scale junction trees, but also provided orders-of-magnitude reduction in query processing times. Further, we developed algorithms to keep INDSEP up-to-date in response to updates to the database.
- We considered a specific class of correlated probabilistic data called *Markovian streams*. Markovian streams, which constitute a large class of naturally occurring correlated probabilistic data, have a repeated cor-

relation structure. We show how to exploit the structured nature of correlations in such sequences, which enables us to build an efficient query processing architecture. We also developed incremental query operator algorithms that can reuse the previous computation during query processing.

- Finally, we proposed an alternative query evaluation model for probabilistic databases that also provides information about *explanations* for query answers and *sensitivity analysis*. The current query evaluation model in probabilistic databases provides very little intuition to the user about the query results. Existing systems assume query processing over probabilistic database queries as a one-shot process. However, probabilistic databases need to be designed as an interactive application in which users have flexibility to identify relevant input probabilities for a given query and re-evaluate the query with the new values for the probabilities of these tuples. We extend a probabilistic database system to support *sensitivity analysis* and *explanation* analysis. Providing such functionality enables a robust framework for query evaluation in probabilistic databases.

# Bibliography

- [1] E. Adar and C. Re. Managing uncertainty in social networks. *IEEE Data Eng. Bull.*, 30(2):15–22, 2007.
- [2] I. F. Akyildiz and I. H. Kasimoglu. Wireless sensor and actor networks: research challenges. In *Ad Hoc Networks*, 2004.
- [3] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.
- [4] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, 2008.
- [5] L. Antova, C. Koch, and D. Olteanu. From complete to incomplete information and back. In *SIGMOD*, 2007.
- [6] The Apache Derby Project. Web Site. <http://db.apache.org/derby/>.
- [7] S. Arnborg. Efficient algorithms for combinatorial problems with bounded decomposability - a survey. *BIT*, 25(1), 1985.
- [8] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2), 1987.
- [9] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE TKDE*, 4(5):487–502, 1992.
- [10] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3), 1983.
- [11] A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for maintaining chordality. *Discrete Mathematics*, 2006.
- [12] H. C. Bravo and R. Ramakrishnan. Optimizing mpf queries: decision support and probabilistic inference. In *SIGMOD*, 2007.
- [13] H. Chan and A. Darwiche. Sensitivity analysis in markov networks. In *IJCAI*, pages 1300–1305, 2005.

- [14] R. Cheng, J. Chen, and X. Xie. Cleaning uncertain data with quality guarantees. *PVLDB*, 2008.
- [15] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.
- [16] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.
- [17] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB*, 2004.
- [18] H. D. Chon, D. Agrawal, and A. E. Abbadi. Query processing for moving objects with space-time grid storage model. In *Third International Conference on Mobile Data Management*, 2002.
- [19] 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.
- [20] M. Collins. A new statistical parser based on bigram lexical dependencies. In *ACL*, pages 184–191, 1996.
- [21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
- [22] G. Cormode and M. N. Garofalakis. Sketching probabilistic data streams. In *SIGMOD*, pages 281–292, 2007.
- [23] D. G. Corneil, Y. Perl, and L. K. Stewart. A linear recognition algorithm for cographs. *SIAM J. Comput.*, 14(4):926–934, 1985.
- [24] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhater. *Probabilistic Networks and Expert Systems*. Springer, 1999.
- [25] P. Dagum and M. Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artif. Intell.*, 60(1):141–153, 1993.
- [26] N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *PODS*, 2007.
- [27] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

- [28] A. Darwiche and M. Hopkins. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In *ECSQARU*, 2001.
- [29] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications (Third Edition)*. Springer-Verlag, 2008.
- [30] R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *IJCAI*, pages 1319–1325, 2005.
- [31] R. de Salvo Braz, E. Amir, and D. Roth. Mpe and partial inversion in lifted probabilistic variable elimination. In *AAAI*, 2006.
- [32] R. Dechter. *Constraint Networks (Survey)*. John Wiley & Sons, 1992.
- [33] R. Dechter and I. Rish. Mini-buckets: A general scheme for bounded inference. *J. ACM*, 50(2):107–153, 2003.
- [34] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.
- [35] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84, 2006.
- [36] X. L. Dong, A. Y. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, 2007.
- [37] A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo methods in practice*. Springer, 2005.
- [38] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge Univ Press, 1999.
- [39] A. Ferrara, G. Pan, and M. Y. Vardi. Treewidth in verification: Local vs. global. In *LPAR*, 2005.
- [40] J. Finn and J. Frank. Optimal junction trees. In *UAI*, 1994.
- [41] D. Forsyth and J. Ponce. *Computer Vision*. Prentice Hall, 2003.
- [42] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.
- [43] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE PAMI*, 1984.

- [44] L. Getoor. Learning probabilistic relational models. In *SARA*, pages 322–323, 2000.
- [45] M. C. Golumbic, A. Mintz, and U. Rotics. Factoring and recognition of read-once functions using cographs and normality. In *DAC*, 2001.
- [46] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, 2006.
- [47] R. I. H. Corrada Bravo. Model-based quality assessment and base-calling for second-generation sequencing data. *Biometrics*, 2009.
- [48] A. Y. Halevy. Learning about data integration challenges from day one. *SIGMOD Rec.*, 32(3):16–17, 2003.
- [49] J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach - part ii: Explanations. In *IJCAI*, 2001.
- [50] J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach: Part 1: Causes. In *UAI*, 2001.
- [51] E. Hoke, J. Sun, and C. Faloutsos. Intemon: Intelligent system monitoring on large clusters. In *VLDB*, 2006.
- [52] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *Int. J. Approx. Reasoning*, 1996.
- [53] T. S. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *SODA*, pages 346–355, 2007.
- [54] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1), 2006.
- [55] T. S. Jayram, A. McGregor, S. Muthukrishnan, and E. Vee. Estimating statistical aggregates on probabilistic data streams. In *PODS*, pages 243–252, 2007.
- [56] M. I. Jordan. *Learning in Graphical Models (ed)*. MIT Press, 1998.
- [57] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *Proceedings of ACM MOBICOM*, Seattle, WA, August 1999.
- [58] B. Kanagal and A. Deshpande. Efficient query evaluation on temporally correlated probabilistic streams. Technical Report CS-TR-4916, University of Maryland.

- [59] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. Technical Report CS-TR-4867, Univ. of Maryland, 2007.
- [60] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *ICDE*, 2008.
- [61] B. Kanagal and A. Deshpande. Efficient query evaluation over temporally correlated probabilistic streams. In *ICDE*, 2009.
- [62] B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *SIGMOD*, 2009.
- [63] P. C. Kanellakis and S. A. Smolka. Ccs expressions, finite state processes, and three problems of equivalence. In *PODC*, 1983.
- [64] U. Kjaerulff. Triangulation of graphs — algorithms giving small total state space. Technical Report R-90-09, Aalborg University, 1990.
- [65] U. Kjærulff and L. C. van der Gaag. Making sensitivity analysis computationally efficient. In *UAI*, 2000.
- [66] C. Koch and D. Olteanu. Conditioning probabilistic databases. *PVLDB*, 2008.
- [67] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli. On-line fault detection of sensor measurements. *IEEE Sensors*, 2003.
- [68] A. Krause and C. Guestrin. Optimal nonmyopic value of information in graphical models - efficient algorithms and theoretical limits. In *IJCAI*, 2005.
- [69] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM J. Comput.*, 1977.
- [70] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *ACM TODS*, 22(3), 1997.
- [71] J. Letchner, C. Re, M. Balazinska, and M. Philipose. Access methods for markovian streams. In *ICDE*, 2009.
- [72] B. Levy, A. Benveniste, and R. Nikoukhah. High-level primitives for recursive maximum likelihood estimation. *IEEE Trans. on Automatic Control*, AC-41(8), 1996.



- [73] D. Lymberopoulos, A. Ogale, A. Savvides, and Y. Aloimonos. A sensory grammar for inferring behaviors in sensor networks. In *IPSN*, 2006.
- [74] S. Madden. Intel lab data, 2004. <http://berkeley.intel-research.net/labdata>.
- [75] A. M. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, 2002.
- [76] R. Mateescu and R. Dechter. And/or cutset conditioning. In *IJCAI*, 2005.
- [77] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. In *PVLDB*, 2011.
- [78] E. Michelakis, R. Krishnamurthy, P. J. Haas, and S. Vaithyanathan. Uncertainty management in rule-based information extraction systems. In *SIGMOD*, 2009.
- [79] V. Mihajlovic and M. Petkovic. Dynamic bayesian networks: A state of the art. University of Twente Document Repository 2001.
- [80] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learnig*. PhD thesis, UC Berkeley, 2002.
- [81] D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, 2010.
- [82] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6), 1987.
- [83] D. Patterson, L. Liao, D. Fox, and H. Kautz. Inferring high level behavior from low level sensors. In *UBICOMP*, 2003.
- [84] D. J. Patterson, L. Liao, D. Fox, and H. A. Kautz. Inferring high-level behavior from low-level sensors. In A. K. Dey, A. Schmidt, and J. F. McCarthy, editors, *UbiComp*, volume 2864 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2003.
- [85] D. J. Patterson, L. Liao, K. Gajos, M. Collier, N. Livic, K. Olson, S. Wang, D. Fox, , and H. Kautz. Opportunity knocks: a system to provide cognitive assistance with transportation services. In *Sixth International Conference on Ubiquitous Computing, Nottingham, England*, 2004.

- [86] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.
- [87] D. Poole. First-order probabilistic inference. In *IJCAI*, pages 985–991, 2003.
- [88] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *77:257–286*, 1989.
- [89] K. Raptopoulou, M. Vassilakopoulos, and Y. Manolopoulos. Towards quadtree-based moving objects databases. *Lecture Notes in Computer Science, Volume 3255, Jan 2004*, 2004.
- [90] C. Re, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.
- [91] C. Re, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, 2008.
- [92] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *PVLDB*, 2008.
- [93] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2), 2006.
- [94] N. Robertson and P. D. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1), 1984.
- [95] A. D. Sarma, O. Benjelloun, A. Y. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.
- [96] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.
- [97] P. Sen, A. Deshpande, and L. Getoor. Exploiting shared correlations in probabilistic databases. In *VLDB*, 2008.
- [98] P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. In *PVLDB*, 2010.
- [99] P. Senellart and G. Gottlob. On the complexity of deriving schema mappings from database instances. In *PODS*, pages 23–32, 2008.
- [100] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *ICDE*, 1995.
- [101] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. E. Hambrusch. Indexing uncertain categorical data. In *ICDE*, 2007.

- [102] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *VLDB*, 2005.
- [103] The RFID Ecosystem, University of Washington. <http://rfid.cs.washington.edu/>.
- [104] G. Trajcevski. Probabilistic range queries in moving objects databases with uncertainty. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 39–45, New York, NY, USA, 2003. ACM Press.
- [105] G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, 2004.
- [106] T. Tran, C. Sutton, R. Cocci, Y. Nie, Y. Diao, and P. J. Shenoy. Probabilistic inference over rfid streams in mobile environments. In *ICDE*, pages 1096–1107, 2009.
- [107] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3), 1979.
- [108] D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 2008.
- [109] G. Welch and G. Bishop. An introduction to the Kalman filter. <http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html>, 2002.
- [110] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.
- [111] M. Yannakakis. Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, 1981.
- [112] X. J. 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.
- [113] N. L. Zhang and D. Poole. Exploiting causal independence in bayesian network inference. *J. Artif. Intell. Res. (JAIR)*, 5, 1996.