Indexing Correlated Probabilistic Databases

Bhargav Kanagal
bhargav@cs.umd.edu

Amol Deshpande
amol@cs.umd.edu

Dept. of Computer Science, University of Maryland, College Park MD 20742

ABSTRACT

With large amounts of correlated probabilistic data being generated in a wide range of application domains including sensor networks, information extraction, event detection etc., effectively managing and querying them has become an important research direction. While there is an exhaustive body of literature on querying independent probabilistic data, supporting efficient queries over large-scale, correlated databases remains a challenge. In this paper, we develop efficient data structures and indexes for supporting inference and decision support queries over such databases. Our proposed hierarchical data structure is suitable both for in-memory and disk-resident databases. We represent the correlations in the probabilistic database using a *junction tree* over the tuple-existence or attribute-value random variables, and use *tree partitioning* techniques to build an index structure over it. We show how to efficiently answer inference and aggregation queries using such an index, resulting in orders of magnitude performance benefits in most cases. In addition, we develop novel algorithms for efficiently keeping the index structure up-to-date as changes (inserts, updates) are made to the probabilistic database. We present a comprehensive experimental study illustrating the benefits of our approach to query processing in probabilistic databases.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; H.2.4 [**Database Management**]: Physical Design; G.3 [**Mathematics of Computing**]: Probability and Statistics

General Terms

Algorithms, Design, Management, Performance

Keywords

Probabilistic Databases, Indexing, Junction Trees, Caching, Inference queries

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

1. INTRODUCTION

Large amounts of correlated probabilistic data are being generated in a variety of application domains including sensor networks [14, 21], information extraction [20, 17], data integration [1, 15], activity recognition [28], and RFID stream analysis [29, 26]. The correlations exhibited by these data sources can range from simple mutual exclusion dependencies, to complex dependencies typically captured using *Bayesian or Markov networks* [30] or through *constraints* on the possible values that the tuple attributes can take [19, 12]. Although several approaches have been proposed for representing complex correlations and for querying over correlated databases, the rapidly increasing scale of such databases has raised unique challenges that have not been addressed before. We illustrate the challenges and motivate our approach by considering two application domains.

Event Monitoring:
Consider an RFID-based event monitoring application [29, 26] (Figure 1) that detects the occurrence of different types of events based on the data acquired from the RFID readers. Since the raw RFID data is noisy and incomplete, it is typically subjected to probabilistic modeling which results in generation of uncertain events, associated with *occurrence probabilities*. For instance, the event `entered(Mary, conf-room, 2:10pm)` may be assigned a 0.6 probability of actually having occurred (Figure 1). The 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)`, whereas the events `entered(Bob, lounge, 2pm)` and `entered(Bob, conf-room, 2pm)` must be mutually exclusive. 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. The nature of the correlation itself can be quantified by associating probability distributions or constraints over the corresponding events.

Given such data, an application or a user may ask queries such as "how many business meetings occurred over the last week?" (*aggregate queries*) or "what is the likelihood that Bob and Mary attended a meeting given that John did not attend?" (*what-if queries*). As has been observed in much work before [30, 22, 26, 23], ignoring the correlations can result in highly inaccurate results to such queries[1].

[1]For an aggregate query, although the expected value can

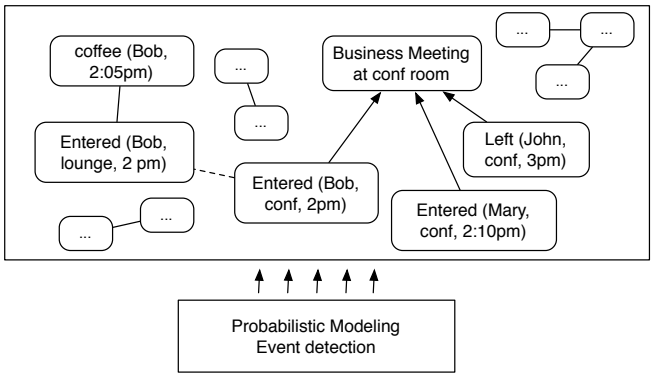


Figure 1: RFID Event monitoring application

Information Extraction (IE):
In an information extraction application [20, 17], a machine learning algorithm automatically extracts structured information from unstructured documents. A simple example of IE is the identification of *relationships*, e.g., person X works for company B ($works(X, B)$), from unstructured text such as "Mr. X left Company A to join Company B". Naturally, an algorithm for this cannot be accurate all the time, hence probabilities are attached to the relations detected. A number of mutual exclusion based dependencies arise in this application. For instance both the tuples works(X,A) and works(X,B) cannot occur simultaneously. A number of interesting queries may be posed on the relations generated, e.g., "where does X work ?" or "how many employees work in Company B ?"

In recent years, several approaches have been developed for managing probabilistic data and for answering queries over them (see, e.g., [10, 35, 1, 30, 31, 3, 2, 34]). The methods proposed in that work can be used to both manage tuple- or attribute-level uncertainties, and to process a rich class of probabilistic queries over them. While some of that work (e.g., [25, 30, 2]) has addressed the issues in representing and querying over complex correlations, their proposed techniques are not scalable to large datasets.

The key challenge in evaluating queries over large-scale correlated databases is that, simple queries involving a few tuple or attribute variables may require accessing and manipulating the probability distributions in the entire database. This is because even if two variables are not directly correlated with each other, they may be indirectly correlated through a chain of other variables in the database. In Figure 1, the events `entered(Bob, lounge, 2pm)` and `left(John, conf-room, 3pm)` are thus correlated, and hence a query involving those two variables must process the correlations among many other variables.

In this paper, we address the challenges in efficiently executing different types of queries over correlated databases by designing novel correlation-aware index data structures. We focus on three types of queries, all of which require reasoning about the correlations: (1) *inference* (what-if) queries, where we are asked to compute a conditional probability distribution over a (typically small) subset of the variables in the database given the values of another subset of variables (where a variable may correspond to the existence of an uncertain tuple or the value of an uncertain attribute); (2) *aggregate* queries, where the desired output is a probability distribution over the aggregate value (which in turn can be used for answering *decision support* queries); (3) *extraction* queries, where the goal is to extract the correlations over a subset of the variables.

Our proposed data structure, called *INDSEP*, builds upon the well-known *junction tree* framework, designed to answer inference queries over large-scale probabilistic graphical models (PGM). We utilize the equivalence between probabilistic databases and probabilistic graphical models [30, 9] for this purpose, by first building a junction tree over the variables in the database. Although such a junction tree over the probabilistic database can be adapted to answer inference queries (as we will discuss in more detail later), this naive approach can not avoid the problem mentioned above, 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. The key contributions of our work can be summarized as:

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 various types of queries efficiently using such a data structure.

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 that illustrate the order-of-magnitude performance benefits of our data structure.

Outline:
The rest of the paper is organized as follows. We provide a brief overview of junction trees and existing algorithms for query processing over them in Section 3. In Section 4, we provide details about our novel index data structure and shortcut potentials. In Section 5, we discuss how we use our index for query processing and in Section 6, we explain how we handle updates. We conclude with our experimental evaluation in Section 7.

often be computed efficiently, computing a *probability distribution* over the result requires reasoning about correlations.

2. RELATED WORK

Indexes for Probabilistic Databases:
Perhaps the most closely related work to ours is the recent work on indexing Markovian streams by Letchner et al. [26]. The authors exploit the restricted correlation structure exhibited by such streams to design a *Markov chain index*, and show how to efficiently execute pattern identification queries. 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). Several works [6, 33, 32] have developed indexing techniques for probabilistic databases, based on R-trees and inverted indices, for efficient execution of *nearest neighbor* queries and *probabilistic threshold queries*. However, these techniques typically assume independence between different data tuples, and focus on a different query workload.

Inference in Graphical Models:
Efficiently evaluating inference queries has been a major research area in the probabilistic reasoning community for many years. A number of exact (e.g., junction trees [16], variable elimination [8], cutset conditioning [27]) and approximate (e.g., loopy belief propagation [36]) techniques have been developed for this problem. Our work builds upon some of this work (in particular junction trees), and we expect our techniques to be useful in answering inference queries over large-scale graphical models as well. Recently Darwiche et al. [11] proposed a data structure called DTree that has superficial similarities to our approach. A DTree specifies a recursive decomposition of a probabilistic graphical model (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., $p(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n)$). 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). We are planning to investigate how to combine our approach with a DTree-like approach in future work.

We note that our approach inherits the limitations of the junction tree approach (which is an exact inference approach, and hence may be infeasible for arbitrary graphical models). Extending our techniques to approximate query answering in such cases (perhaps using the recently developed approximation approach by Choi et al. [7]) is a rich area of further work.

Scalable Inference using Relational Databases:
Bravo et al. [5] address the problem of evaluating inference queries (also called MPF queries) using relational database techniques. They represent each conditional probability distribution as a separate relation, and show how to scalably evaluate inference queries using relational operators. However, that approach is only suitable when the number of variables is small, but the conditional probability distributions (that quantify the correlations) are large; we address the complementary problem where the number of variables is very large, but the probability distributions are relatively small.

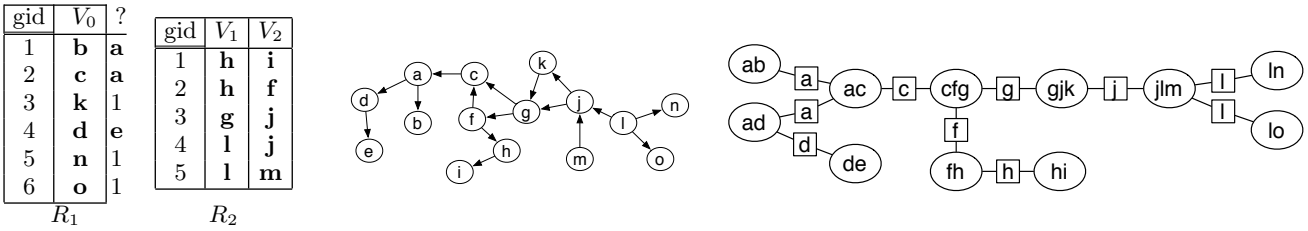
3. PRELIMINARIES

We begin with briefly presenting the relevant background on probabilistic databases and their equivalent representation as PGMs. In addition, we describe the junction tree representation of a PGM, and discuss how to execute various types of queries using junction trees.

Probabilistic Graphical Models (PGMs):
PGMs comprise a powerful class of approaches that enable us to compactly represent and efficiently reason about very large joint probability distributions [8]. A PGM is typically represented using a directed or an undirected graph, in which the nodes represent random variables and the edges represent the direct dependencies/correlations between the random variables. Figure 2(b) depicts a directed PGM on a set of random variables $\{a, b, \ldots, o\}$. 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(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(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 all the probability distributions in the PGM. Missing edges encode the conditional independences between the random variables. For example, in Figure 2(b), e is independent of a if we know the value of the random variable d.

Probabilistic Databases as PGMs:
A probabilistic database may exhibit either *tuple-existence uncertainty*, where the existence of a tuple in the database is uncertain, or *attribute-value uncertainty*, where the value of an attribute is not known for sure, or a combination of both. Further, there may be complex correlations present in the database at various levels and possibly across relations. All of these can be modeled in a generic manner using PGMs by introducing appropriate random variables and joint probability distributions over them [30, 13]. Tuple existence uncertainty can be modeled by introducing a binary random variable in the PGM, which takes value 1 if the tuple exists and 0 otherwise. Attribute uncertainty can be modeled by using a random variable to denote the value of the attribute. Any correlations are then captured by adding appropriate edges to the model, and by quantifying each node with the probability distribution that specifies how the value of the node is dependent on those of its parents.

Figure 2(a) shows a probabilistic database on two event relations R_1 and R_2. R_1 has both tuple uncertainty and attribute uncertainty, while R_2 only exhibits attribute uncertainty. For example, the attribute V_0 in relation R_1 is probabilistic, hence in tuple $(2, c)$, c is a random variable that denotes the value of V_0. Also the tuple belongs to the database whenever the random variable a takes a value 1. Notice the correlation between tuples 1 and 2 in R_1, i.e., either both of them belong to the database together, or none



(a) Probabilistic Database (b) PGM representation (c) Junction Tree for the PGM in (b)

Figure 2: Running 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.

of them appear, depending on the value of the random variable a. Further, many more complex correlations can occur among the random variables. Figure 2(b) shows one possible set of correlations among the tuples in the database using a PGM.

Given such a database with uncertainties captured using a PGM, query execution can be seen as equivalent to inference on an appropriately modified PGM (by adding new random variables corresponding to the intermediate tuples generated during execution) as shown in Sen et al. [30].

Junction Tree Representation of PGMs:
A PGM can be equivalently described using a *junction tree* representation [16], also commonly known as a *clique tree* in the graph theory literature.

Due to space constraints, we omit the full details involved in the construction of a junction tree, and only discuss the key properties of the junction tree and the algorithms used for evaluating queries over junction trees. Briefly speaking, we must first convert the directed graph corresponding to the PGM into an undirected graph by *moralizing* the graph (adding edges between all pairs of parents of a node if they are not already connected) and by making all the edges undirected. Then the graph is *triangulated* by possibly adding further edges, and the junction tree is built over the maximal cliques of the resulting graph.

In a junction tree, there are two types of nodes, *clique* nodes and *separator* nodes. The clique nodes in the junction tree correspond to the *maximal* cliques in the undirected PGM and the separator nodes correspond to the cut vertex sets that separate the maximal cliques in the 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 [18] 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. 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:

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

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(b), the value of the joint distribution is:

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

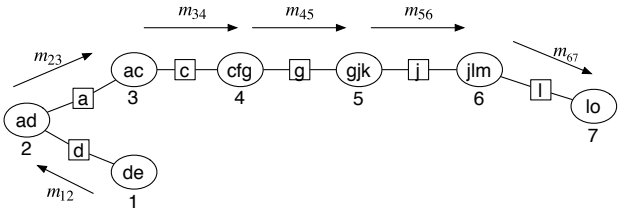


Figure 3: Path constructed for query {e,o}

Query Processing:
There are three main types of queries that we may be interested in executing over a probabilistic database.

Extraction queries:
An extraction query is specified by a set of variables. The output of the query 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(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 3. 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: An inference query is specified by a set of variables and the result of the 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 Hugin's algorithm [8] 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:

$$p(g, k) = \Sigma_j \; p(g, j, k)$$

Now consider the inference query {e,o}. As before, we run the extraction query and obtain the path shown in Figure 3. 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 3. In the first step, clique de sends a message m_{12} (See Figure 3) to clique ad which is basically the value of the joint distribution $p(d, e)$. After receiving this messgae, 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: Aggregation queries are specified using a set of random variables and the aggregation function, such as `SUM`, `MIN`, `MAX` etc. 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. [30] for computing the probability distribution of the aggregate value. We discuss aggregate query evaluation in more detail in Section 5.2.

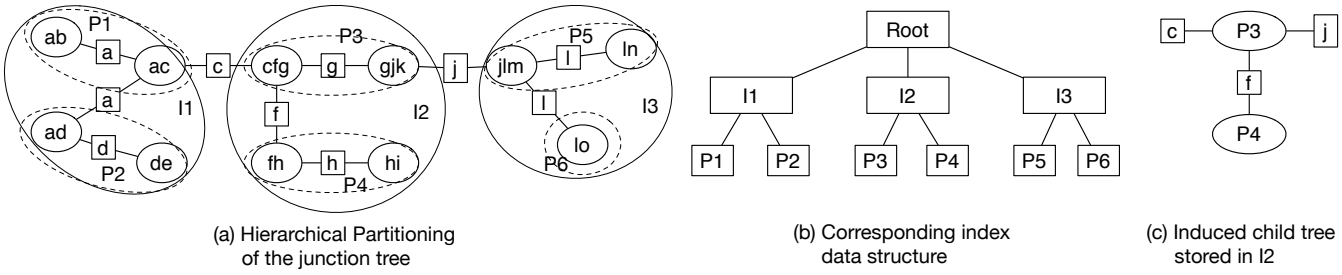
As illustrated above, we reduce the problem of query processing on probabilistic databases to the problem of Steiner tree computation on trees (Since all 3 queries require to perform the extraction query first). However, the above algorithms do not scale for very large junction trees mainly for the following reasons. Firstly, for very large junction trees, even searching for the cliques in the tree is expensive. Secondly, as shown for the inference query {e,o}, the size of extracted junction tree can be very large, i.e., almost as big as the underlying junction tree itself. To counter these issues, we develop index data structures for faster searching, and augment the index with shortcut potentials to reduce the size of the extracted junction trees. We begin with describing our proposed indexing data structure.

4. 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.

4.1 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(c) as shown in Figure 4(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 4(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.



(a) Hierarchical Partitioning of the junction tree

(b) Corresponding index data structure

(c) Induced child tree stored in I2

Figure 4: (a) shows a hierarchical partition of the junction tree shown in Figure 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).

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 4(c).
- **(C5)** Set of *shortcut potentials* corresponding to the children of this node. We describe shortcut potentials in Section 4.2.

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 4(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 4(b) stores the induced tree shown in Figure 4(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.

4.2 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, which represents a path connecting the variables X and Y in a junction tree. As shown earlier (Section 3), we can compute p(X,Y) using the following sequence of messages from the clique C_1 towards C_3.

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

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 access to the joint distribution $p(C, D)$ stored in the above example, the computation can be faster: we would

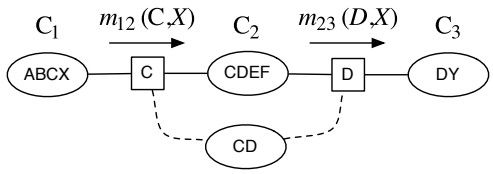


Figure 5: Illustrating overlays and shortcut potentials. Using the cached potential $p(C, D)$ allows us to shortcut the clique C_2 completely.

replace the computation of the message m_{23} with the equation shown below.

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

$p(C, D)$ 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 I completely.
For example, in Figure 4(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 $\{s_1, s_2, \ldots s_k\}$. 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 6, we describe how to update shortcut potentials efficiently when updates occur to the database.

4.3 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).

4.3.1 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. [24].

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. [24] 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 4.2) 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.

4.3.2 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 4.1) 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.

4.3.3 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. 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.

5.1 Inference/Extraction Queries

As illustrated earlier (Section 3), inference queries can be solved by constructing a tree joining all the query variables and then running Hugin's 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 ∀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) = φ & ∃ shortcut P s.t. nrs ∈ P then
15:         gm.add(I.shortcutpotential(nrs))
16:      else
17:         gm.add(query(I, I(V) ∪ 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(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 6(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 6(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 6(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 6(d). Notice that the graphical model is much smaller than the one shown in Figure 3 (which was constructed without the index).

5.2 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 $S = \{s_1, s_2, \ldots s_n\}$, if it satisfies the following condition.

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

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 [22], we showed how to exploit decomposability of aggregates to efficiently execute aggregation queries for the special case of Markov 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 extraction 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(a). This corresponds to computing the aggregate of the random variables {b,c,d,k,n,o}. In the naive method (Section 3), 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:

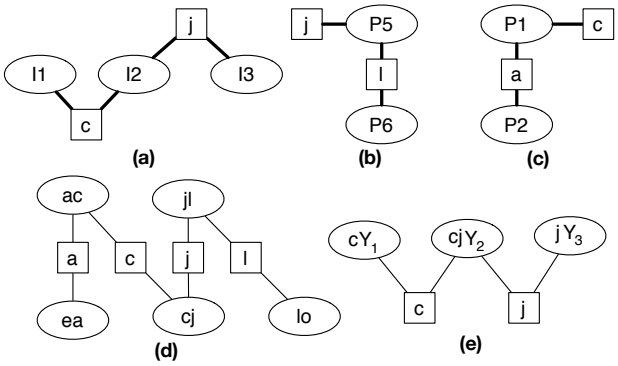


Figure 6: The Steiner trees generated at different index nodes while executing the inference query {e,o} on the junction tree in Figure 2(c) is shown in (a), (b), (c). (d) shows the final graphical model generated as a result of the extraction query {e,o}. The junction tree generated by the aggregation query is shown in (e)

{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 agg-inf(I_1,{b,c,d},{c}). Now, this induces further recursive calls on the partitions agg-inf(P_1, {b,c}, {c,a}) and 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 agg-inf(P_3, {k}, {c,j}). Similarly the recursive call on I_3 leads to two inference queries agg-inf(P_5, {n}, {j,l}) and 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 6(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.

6. 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.

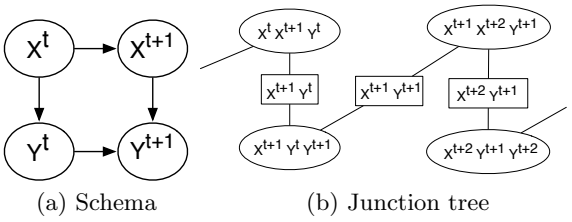
- The first is a modification of the existing data, i.e., modification of a probability distribution, or the assignment of a deterministic value to an existing random variable. For instance, if we verify the occurrence of the tuple with *gid=2* in the probabilistic database of Figure 2(a), then we need 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.

6.1 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 Hugin's algorithm (Section 3). 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.



(a) Schema (b) Junction tree

Figure 7: We generate the Markov sequence database using the schema shown in (a). The junction tree structure of the Markov sequence is shown in (b)

6.2 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 $S = \{s_1, s_2, \ldots, s_k\}$, 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 an new id to this variable results in the violation of the contiguous variable name property (See Section 4), 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. [4]. 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.

6.3 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 deletion.

7. EXPERIMENTAL EVALUATION

In this section, we present a comprehensive experimental evaluation using a prototype system that we have built. Our results show that using our proposed index can result in orders of magnitude performance improvements compared to the prior approaches in most cases. We also study the key properties of the index structure itself, and the overheads of keeping it up-to-date. We begin with a brief description of our implementation and our experimental setup.

7.1 Implementation Details

We implemented a prototype system using Java that supports the *INDSEP* data structure, querying using the index, and updating the index. We simulated the disk as an array of disk blocks, each of which is a *serialized byte array* of size BLOCK_SIZE (a parameter). Any read, write access to the disk is made via the *disk manager*, a singleton class and a constantly running module, that manages the disk blocks. Each index node (leaf or interior) is stored in a single disk block. To simulate the disk behavior as closely as possible, the data is read and written in the units of block sizes. For example, to add a new clique to a partition P, we first read off the disk block containing P into memory and then add the new clique and write the entire block back into the original location of P. The other software components in the system are the query processor, and the update manager, both of which use the disk manager for accessing the index and the partitions containing the junction tree.

7.2 Experimental Setup

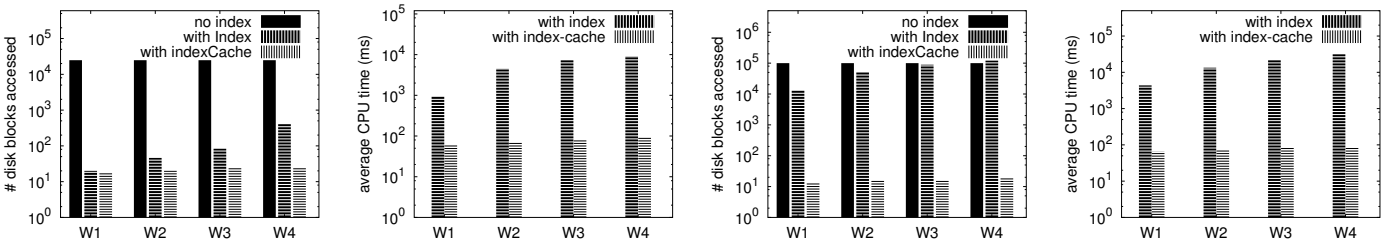
All of our experiments were carried out on a machine with a 2.4 GHz Intel Core 2 Duo processor and 2GB memory. We evaluate the performance of our index on the following two probabilistic databases.

- **General probabilistic database:** We generate a probabilistic database on 2 relations that is representative of a typical event monitoring application (see Section 1). 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.
- **Markov Sequence database:** We generate a Markov sequence database [22] with schema shown in Figure 7(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.

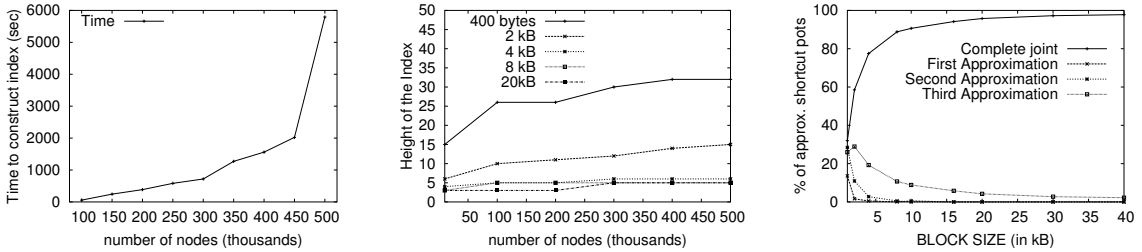
- W_1**:** Shortest-range queries. These are queries that have a span of about 20% of the junction tree.
- W_2**:** Short-range queries. These have a span of 40% of the junction tree.
- W_3**:** Long-range queries. These have a span of 60% of the junction tree.
- W_4**:** Longest-range queries. Each 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



(a) # disk blocks accessed (b) Average CPU time (c) # disk blocks accessed (d) Average CPU time

Figure 8: 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 Markov sequence database. We note that the graph is in *logarithmic scale*, so the gains are substantially more than what is apparent.



(a) Time for bulk loading the index (b) Height of the index as a function of block size (c) % of shortcut potentials that are approximate

Figure 9: 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.

generate 4 different update workloads (for the first probabilistic database). Each newly added variable has a set of 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:

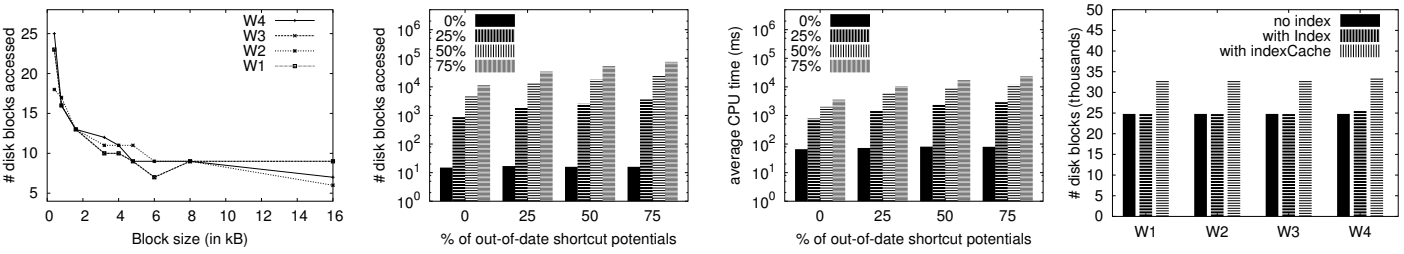
- **No index:** In this case, we do not maintain any indexes in the database and perform query processing using the naive technique described in Section 3.
- **Index without caches:** In this case we maintain the index over the junction tree, but do not maintain any shortcut potentials. The key advantage 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.

7.3 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 8(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 8(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 8(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 Fig-



(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 10: 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.

ure 9(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 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 9(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 9(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 10(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 Markov 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 10(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 8(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 10(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.

8. CONCLUSION

In recent years, there has been an increase in the amount of uncertain probabilistic data which is correlated both spatially and temporally owing primarily to the proliferation of sensor networks and other measurement infrastructure and the increasing use of machine learning techniques for processing their data. Developing scalable query processing techniques over such data has become an important task in database research. In this paper, we developed an index data structure for correlated probabilistic databases which allows for efficient processing of decision support queries – including inference queries and aggregation queries. Our techniques are based on tree partitioning algorithms that enables us to hierarchically partition the junction tree corresponding to the probabilistic database, thereby generating an n-ary search tree data structure. In addition, we introduce novel shortcut potentials that further reduce query processing time by orders of magnitude. We also develop efficient techniques for keeping the index up-to-date in re-

sponse to updates. Our experimental results demonstrate the benefits of our indexing mechanisms for query processing in probabilistic databases. We are currently in the process of integrating the INDSEP data structure with a full-fledged probabilistic database system that supports specifying and manipulating correlations using a declarative language.

Our work so far has raised many interesting research challenges that we are planning to pursue in future. The *INDSEP* data structure, since it is based on the junction tree approach (an exact inference technique), inherits the limitations of that approach, and is not feasible for performing inference on models with large treewidths. One of the interesting questions is whether we can incorporate an approximate inference technique (like loopy belief propagation) into the index structure. Another option to handle large treewidth models is to approximate the model by systematically reducing its treewidth [7]. Other research directions we are planning to pursue include sharing computation between multiple inference queries, and developing more elegant techniques for handling deletions.

Acknowledgement

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

9. REFERENCES

[1] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.

[2] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, 2008.

[3] L. Antova, C. Koch, and D. Olteanu. From complete to incomplete information and back. In *SIGMOD*, 2007.

[4] A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for maintaining chordality. *Discrete Mathematics*, 2006.

[5] H. C. Bravo and R. Ramakrishnan. Optimizing mpf queries: decision support and probabilistic inference. In *SIGMOD*, 2007.

[6] 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.

[7] A. Choi and A. Darwiche. Focusing generalizations of belief propagation on targeted queries. In *AAAI*, 2008.

[8] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhater. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[9] N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *PODS*, 2007.

[10] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

[11] A. Darwiche and M. Hopkins. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In *ECSQARU*, 2001.

[12] R. Dechter. *Constraint Networks (Survey)*. John Wiley & Sons, 1992.

[13] A. Deshpande, L. Getoor, and P. Sen. *Graphical Models for Uncertain Data*. Managing and Mining Uncertain Data. Charu Aggarwal ed., Springer, 2009.

[14] A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *CIDR*, 2005.

[15] X. L. Dong, A. Y. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, 2007.

[16] J. Finn and J. Frank. Optimal junction trees. In *UAI*, 1994.

[17] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, 2006.

[18] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *Int. J. Approx. Reasoning*, 1996.

[19] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *JACM*, 31(4), 1984.

[20] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1), 2006.

[21] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *ICDE*, 2008.

[22] B. Kanagal and A. Deshpande. Efficient query evaluation over temporally correlated probabilistic streams. In *ICDE*, 2009.

[23] C. Koch and D. Olteanu. Conditioning probabilistic databases. *PVLDB*, 2008.

[24] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM J. Comput.*, 1977.

[25] L. V. S. Lakshmanan et al. Probview: a flexible probabilistic database system. *ACM TODS*, 1997.

[26] J. Letchner, C. Re, M. Balazinska, and M. Philipose. Access methods for markovian streams. In *ICDE*, 2009.

[27] R. Mateescu and R. Dechter. And/or cutset conditioning. In *IJCAI*, 2005.

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

[29] C. Re, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, 2008.

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

[31] P. Sen, A. Deshpande, and L. Getoor. Exploiting shared correlations in probabilistic databases. In *VLDB*, 2008.

[32] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. E. Hambrusch. Indexing uncertain categorical data. In *ICDE*, 2007.

[33] Y. Tao et al. Indexing multi-dimensional uncertain data with arbitrary probability densityfunctions. In *VLDB*, 2005.

[34] D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 2008.

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

[36] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Generalized belief propagation. In *NIPS*, 2000.