Efficient Querying of Correlated Uncertain Data with Cached Results

Jinchuan Chen[1], Min Zhang[2], Xike Xie[3], and Xiaoyong Du[1,2]

[1] Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, China
{jcchen,duyong}@ruc.edu.cn

[2] School of Information, Renmin University of China
zhangmin0453@ruc.edu.cn

[3] Department of Computer Science, Aalborg University, Denmark
xkxie@cs.aau.dk

Abstract. Although there have been many efforts for management of uncertain data, evaluating probabilistic inference queries, a known NP-hard problem, is still a big challenge, especially for querying data with highly correlations. The state-of-art exact algorithms for accelerating the evaluation of inference queries are based on special indices. Besides, with the observation of the existence of many frequent queries, some researchers try to improve efficiency by reusing previously queried results. Indexing depends on the static properties like data distributions, whereas caching is in favor of the dynamic features like query workload. In this paper we propose a new approach for speeding up the evaluation of inference queries by caching frequent results in a junction tree-based hierarchical index. To the best of our knowledge, this is the first effort on utilizing both the static (data) and dynamic (query workload) properties to efficiently evaluate probabilistic inference queries. Moreover, according to our experience, different caching strategies may significantly affect the query performance. Basically a good caching strategy needs to have high cache hit ratio with limited space budget.Based on these considerations, we propose a novel caching approach, called *FVEC*, and present corresponding algorithms for efficiently querying correlated uncertain data. We further conduct a series of extensive experiments on large uncertain datasets in order to illustrate the effectiveness and efficiency of our proposed approaches. As illustrated by the results, compared with previous solutions, our method could greatly improve the query performance.

1 Introduction

Recently data uncertainty becomes an intrinsic property in many applications. For example, in an information integration system [6], the automatically generated schema mappings could be imprecise. Hence the data in the integrated database would be uncertain. As another example, when extracting information from unstructured data like web pages, top-k possible results would be generated and the database obtained would be a probabilistic database [18]. Due to its high applicability, the management of uncertain data is attracting more and more research interests during the past decade.

One of the core problems for managing uncertain data is the evaluation of probabilistic queries, a known #P-complete problem [5]. Generally, a probabilistic query is to compute the probabilities of some complex random events based on the joint distribution of all the uncertain data items in a dataset. In probabilistic theory, it is exactly the process of *probabilistic inference*. Specifically, we only discuss the queries of extracting the joint distribution of a set of variables from the whole joint distribution. But our methods could be applicable for general cases. In this paper, we use the two terms *probabilistic query* and *inference query* interchangeably.

Many research works are proposed for solving this problem. Most of these approaches are based on the assumption that uncertain data are either fairly independent or partitioned in independent groups. Kanagal et al. propose a novel index called INDSEP to efficiently query correlated uncertain data [8]. The main idea of INDSEP is to cluster uncertain data into partitions and organize them in a hierarchical structure. When evaluating probabilistic queries, INDSEP can then prune many computation efforts with the hierarchical structure and the pre-computed *shortcut* potentials.

The INDSEP structure is designed as a disk-based index. The main part of the index is stored in disk and will be loaded into main memory in need. The size of each index node is limited by the size of a disk page. However, as found in many projects [3,5,9], the major bottleneck of answering probabilistic queries is not I/O but CPU. Nowadays, it is not that expensive to buy a machine with several gigabytes RAM, but the response time could still be quite long even if the whole INDSEP structure has been loaded into the main memory.

On the other hand, in reality the *Pareto principle* is a common phenomena in many applications. This principle, also known as the 80-20 rule, states that roughly 80% of the queries focus on 20% data items [1]. With this observation, and the availabity of large main memory, an intuitive solution is to cache frequent results and use them for answering queries. In [15], the frequent intermediate results are stored in the main memory in order to reduce response time.

The indecis are built based on the static properties like data distributions, e.g. two variables X and Y would be probably placed in the same partition if they have strong correlations. On the contrary, when deciding which items should be cached, the major concern is the dynamic features like query workload, e.g. we may consider to store the joint distribution of $\{X, Y, Z\}$ if it is contained by many previous queries. Intuitively, we can further improve the query performance by benefiting from both the static (data) and dynamic (workload) features. Our work is motivated by this observation. In brief, our basic idea is to adapt the INDSEP structure to accommodate the frequent accessed results. We also revise the original algorithm for evaluating inference queries on INDSEP so that the cached results could be utilized.

However, this approach seems not to work well if we directly adopt the caching strategy in [15], i.e. storing frequent intermediate results. According to our experimental results, this caching method, called *SubQuery* in this paper, can only improve the query performance by about 10% compared with original INDSEP. The reasons are two folds. Firstly, there are huge number of possible intermediate queries and the cache hit ratio of *SubQuery* cannot be satisfactorily high. Note that an intermediate query is exactly to extract the joint distribution of a set of random variables. The number of possible intermediate queries, or combinations of variables, would be extremely huge when the data set is large. The second reason lies in its high space consumption. Storing a joint distribution requires to store all the entries of this distribution, whose volume will increase exponentially along with the enlargement of the number of variables contained in the distribution. The *SubQuery* approach usually requires to cache sub-queries with relatively large cardinality and consumes lots of space. We will revisit this issue in Sec. 3.1.

With this concern, we propose a new caching technique, called *Frequent-Variables*, which only store the joint distribution of frequent single variables with their corresponding separators. Separators are special variables in junction-tree which are required to *connect* the partial results from different partitions. More details about separators and junction-tree would be discussed in Section 2. Compared with *SubQuery*, the *Frequent-Variables* technique could control the size of cached joint distributions because usually there are only few separators. In this way, we could reduce the memory consumption. The cache hit ratio would also be improved since we now able to reserve more frequent variables.

More importantly, different from [15], we are not only trying to match the intermediate sub-queries with cached results, but also pruning unnecessary intermediate sub-queries and consequentially shortening the evaluation path. Specifically, we find that we can use some cached results for acceleration even if the cached variables are NOT included in a query. Based on this observation, we propose the so-called *Express-Channels* technique which could further save lots of subqueries. Our proposed method integrates these two techniques. Hence we call it *FVEC (Frequent Variables with Express Channels)*.

Our major contributions in this paper are summarized as follows:

- An approach for efficiently querying large volume of uncertain data with high correlations based on cached results.
- Novel caching approaches for storing the requent results.
- Efficient algorithms for evaluating inference queries based on the cached information.
- Extensive experiments on large datasets for verifying the effectiveness and efficiency of our proposed approaches.

Next we will review some issues which our work is based on, such as uncertain data model, the junction tree and the INDSEP structures in Section 2. After that, we will propose our approaches for caching frequent variables in Section 3, and a series of techniques to make good use of cached results to prune subqueries in Section 4. The experimental study will be illustrated in Section 5. Section 6 will discuss related works. We then conclude this paper in Section 7.

2 Background Knowledge

In this section, we briefly review several important issues related to our work. We will first introduce a data model for representing uncertain data.Secondly, we will show the basic idea of junction tree [10], a data structure for efficient processing of inference queries, and explain the structure of INDSEP [8], an index for querying correlated uncertain data.

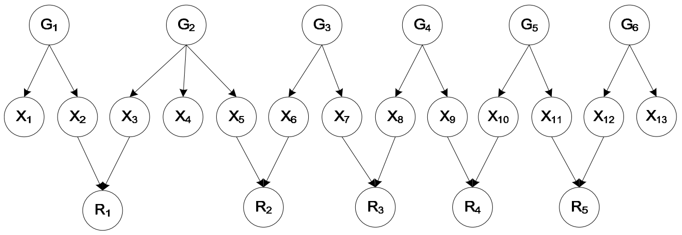

Fig. 1. GRN Representation for a Probabilistic Database

2.1 GRN Model

There have been several models for representation of uncertain data such as attribute uncertainty and tuple uncertainty [14]. But most models are only applicable for data with no correlations. Recently Chen et al. propose a model called *GRN* (**G**enerator-**R**ecognizer **N**etwork) for expressing correlated uncertain data [2]. As our work focuses on querying highly correlated uncertain data, we decide to adopt the GRN model.

The GRN model is a special Bayesian Network. Figure 1 illustrates a simple example for representing a probabilistic database in GRN. Each node is labeled with the random variable to which it is corresponded. Each arrow depicts a dependency among variables and is attached with a conditional probability table (CPT). The nodes labeled by $X_1 \dots X_{13}$ are tuple variables. Basically each tuple in the probabilistic database will have a corresponding node in GRN. The nodes $G_1 \dots G_6$ are called *generators*, which are inserted to express the correlations among tuple variables. For example, with G_1 and the attached CPTs, X_1 and X_2 should be mutually exclusive. Finally, the nodes $R_1 \dots R_5$ are recognizers for representing the correlations among X-tuples, e.g. two groups $\{X_1, X_2\}$ and $\{X_3, X_4, X_5\}$ are correlated with the existence of R_1. .

2.2 Junction Tree and INDSEP

Intuitively, the evaluation efficiency would be much better if we can perform inference on a small part of the whole distribution, which leads to the introduction of junction tree [10]. The nodes in the Bayesian network would be clustered into *cliques*. Each clique has the property that all nodes inside it are pairwise linked. A junction tree will be obtained then by applying an elimination sequence on the triangulated 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 GRN and the separator nodes correspond to the vertex sets that separate the maximal cliques. Suppose χ is the set of all random variables, C_i is denoted as a clique and s_j is denoted as a separator, the overall joint distribution represented by a junction tree can be computed as follows.

$$P(\chi) = \frac{\prod_{C_i} P(C_i)}{\prod_{s_j} P(s_j)} \tag{1}$$

After construction and calibration, the potential of C_i (or s_j) is exactly the marginal distribution of C_i (or s_j).

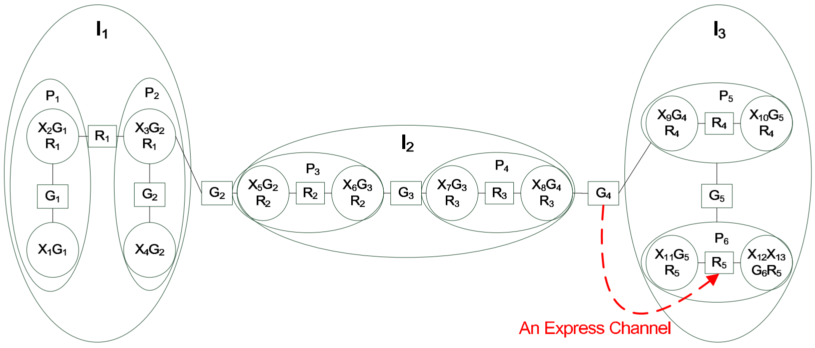

Fig. 2. The Partition Result for the Junction Tree Builit on Figure 1

Kanagal et al. find that we could avoid going into some of these intermediate nodes if the joint distributions of some special variables are pre-computed. With this observation, they propose the so-called INDSEP structure [8], which is to hierarchically partition a junction tree into connected subtrees and subsequently construct the index. Figure 2 shows a hierarchical partition of the junction tree built on the above GRN. Each index node in INDSEP corresponds to a connected subtree of the junction tree, and it has a set of separators which separate this node from its siblings. A separator is basically a set of random variables. The joint distribution of all the separators of a node is called *shortcut* potential and can be used for speeding up query processing.

For completeness, we now briefly summarize the main idea of answering inference queries on INDSEP. When receiving an inference query, INDSEP will first search for the children nodes of the root which contain the queried variables. Next, a so-called Steiner tree [7] will be constructed which connect all the target nodes. For each node in the Steiner tree, a subquery will be issued on it which includes not only the corresponding query variables, but also separators of this node. For the cases where a subquery contains separators only, INDSEP just takes the node's shortcut potential into computation. After obtaining the results of all subqueries, the final result will be obtained by computing the joint distribution of all subquery results and intermediate separators, and marginaling out the redundant variables. Note that this process is recursive, i.e. the evaluation of each subquery will follow the same process before going down to the leaf level. Imagine in an INDSEP containing thousands of nodes, the query performance could be quite low when the queried variables are far away from each other. The performance could still be improved greatly if the results of some queried variables have been cached.

3 Caching Frequent Query Results

In this section, we will illustrate our approaches for caching frequent query results, i.e. *SubQuery*, and *FVEC*. We also give a detailed discussions to explain why they have different performance.

Table 1. Results Cached in Each Node in Example 1

Node	Cache(SubQuery)	Cache(FVEC)
I_1	$\{X_1, X_2, X_4, G_2\}$	$\{X_1, G_2\}, \{X_2, G_2\}, \{X_4, G_2\}$, $\{R_1, G_1, G_2\}$
I_2	$\{X_5, X_6, X_8, G_2, G_4\}$	$\{X_5, G_2, G_4\}$,$\{X_6, G_2, G_4\}$,$\{X_8, G_2, G_4\}$, $\{R_2, G_2, G_4\}$, $\{R_3, G_2, G_4\}$
I_3	$\{X_{12}, G_4\}$	$\{X_{12}, G_4\}$, $\{R_5, G_4\}$
P_1	$\{X_1, X_2, R_1\}$	$\{X_1, R_1\}$, $\{X_2, R_1\}$
P_2	$\{X_4, R_1, G_2\}$	$\{X_4, R_1, G_2\}$
P_3	$\{X_5, X_6, G_2, G_3\}$	$\{X_5, G_2, G_3\}$, $\{X_6, G_2, G_3\}$
P_4	$\{X_8, G_3, G_4\}$	$\{X_8, G_3, G_4\}$
P_6	$\{X_{12}, G_5\}$	$\{X_{12}, G_5\}$

Example 1. For ease of illustration, we now give an example before touching the detailed cache approaches. Suppose we now need to obtain the marginal distribution $P(X_1, X_2, X_4, X_5, X_6, X_8, X_{12})$ over the INDSEP structure shown in Figure 2. The query is first performed on the root node. Since these variables locate in I_1, I_2 and I_3 respectively. The Steiner tree for connecting the two nodes would be $I_1 - I_2 - I_3$. Three sub-queries are then generated for these three nodes, i.e. (X_1, X_2, X_4, G_2) for I_1, $(X_5, X_6, X_8, G_2, G_4)$ for I_2 and (X_{12}, G_4) for I_3. Note that when a children node v is queried, its separator S_v must be included in the query. Otherwise we cannot assembly the final result with the partial ones [8]. This process will be repeated for each of these sub-queries until either the result of a sub-query is found in one shortcut or a leaf node is met.

3.1 The SubQuery Approach

As shown in Example 1, when a query is evaluated over an INDSEP tree, many sub-queries would be generated. Each sub-query may also be broken down into several new children sub-queries. Following [15], the *SubQuery* approach will store the results of frequent sub-queries at the corresponding nodes. For example, since the sub-query (X_1, X_2, X_4, G_2) is queried over I_1, we then store the result $P(X_1, X_2, X_4, G_2)$ at node I_1. Similarly, we will store $P(X_5, X_6, X_8, G_2, G_4)$,

$P(X_{12}, G_4)$ at I_2 and I_3 respectively. In this way, when processing new queries, we can avoid re-evaluate some intermediate queries if they are found in the cache. Table 1 lists the cached data at each node after evaluating the query in Example 1 according to the *Subquery* approach. Moreover, we attach a memory limit on each node, and adopt the classical LRU (Least Recently Used) algorithm for updating a cache when it is full.

Discussions. According to the experimental results, the *SubQuery* approach could really save the computation of many sub-queries and reduce the response time. But the improvement is only about 10% compared with the original IND-SEP without any cache. After careful analysis, we find an interesting phenomenon that most cache hittings appear in low-level nodes. Due to the recursive evaluation process, a sub-query generated in a high-level node usually results in much more sub-queries than one generated in a low-level node. Hence it is more valuable to kill sub-queries in high-level nodes. In our experiments, most killed sub-queries by the *Subquery* approach locate in 4-5 levels (with tree height equals to 5). That is why the performance is only slightly improved.

Why it is so hard to hit high-level caches? The reason lies in the number of sub-queries. Note that a sub-query is exactly a combination of variables, and the number of possible sub-queries would increase very fast when its cardinality enlarges. On average, the cardinality of sub-queries appearing in high levels would be larger than that of sub-queries in low levels. Hence there are large number of possible sub-queries for high-level nodes and the chances of finding a new query in the cache would be very low.

Another shortcoming of the *Subquery* approach is its high memory cost. The data stored in cache are some joint distributions of several variables. The number of entries in a joint distribution would increase exponentially with the number of variables contained in it. For sub-queries with big cardinality, the memory budget would be quickly run out, and we have to perform the cache updating very often. Again, this problem will become much more serious in high-level nodes where sub-queries are quite large. Frequently cache updating causes not only extra time cost, but also low cache hit ratio.

With these observations, we decide to decompose the results of sub-queries into small fragments, i.e. results of single variables. In this way, we are able to accommodate more distributions within the same cache size, and to identify frequent variables. Furthermore, we find that by leveraging some special variables we could build *express channels* between the nodes in high-level and low-level. With these channels, we could skip large parts of the evaluation path and reduce the overall evaluation time. This approach, called *Frequent Variable with Express Channel* (FVEC in short) will be illustrated in the next part.

3.2 The FVEC Approach

The FVEC approach is based on the idea of caching the distributions of frequent variables. However, it is no use to cache the marginal distribution of any single variables. For example, we cannot obtain the joint distribution of $P(X_1, X_2)$

from the cached results $P(X_1)$ and $P(X_2)$ because they are not independent. Specifically, we need to choose some *bridge* variables for a frequent variable according to the *d-separation* theorem [2] and store the joint distribution of the frequent variable and its bridges. In INDSEP, these bridge variables are usually the separators of the node containing the frequent variable. For example, at node I_1 in Fig. 2, we should store $P(X_1, G_2)$ at I_1 when X_1 is found to be frequent since G_2 is the separator of I_1. Similarly, we store $P(X_5, G_2, G_4)$ at node I_2. Therefore, if (X_1, X_5) is queried in the future we could derive it by $P(X_1, G_2) * P(X_5, G_2)/P(G_2)$. Here $P(X_5, G_2)$ could be computed from $P(X_5, G_2, G_4)$.

Definition 1. Frequent-Variable. *Suppose a variable X is found to be frequent in a node v, the Frequent-Variable scheme will store $P(X, S_v)$ at this node. Here S_v is the set of separators of v.*

Now we explain the idea of *express channel*. In Example 1, in order to evaluate the sub-query (X_{12}, G_4), we need to build a long evaluation path from the root node to the leaf node containing X_{12}. During this process, we must process a set of sub-queries, i.e. (G_4, G_5), (G_5, R_5) and (R_5, X_{12}), and obtain the final result based on Equation 1. Suppose next time we want to query X_{13}. We have to go through the same evaluation path and evaluate almost the same set of sub-queries except that the sub-query (R_5, X_{12}) is replaced by (R_5, X_{13}).

The evaluation of X_{13} could be accelerated if we cache the result $P(G_4, R_5)$. In this case, we just have two sub-queries (G_4, R_5) and (R_5, X_{13}). As shown by the dotted red arrow in Fig. 2, this cache scheme is like to build a channel from I_2 to the leaf node. Hence we call it *express channel*. Note that in practical there could be hundreds of thousands of variables and the evaluation path would be much longer. Therefore this method could help us cut off many partial evaluation path and skip many sub-queries.

Definition 2. Express-Channel. *If a node v is accessed, we would store the joint distributions of S_v and S'_v at each ancestor node v' of v.*

The major advantage of the express-channel scheme is the ability of killing many sub-queries in high-level nodes. As aforementioned, with express channels, we can directly pull the variables from leaf nodes. Moreover, note that with this scheme we can accelerate queries with no cached variables, e.g. the above query X_{13}. This is because these channels are linked with separators and these separators will be shared by different queries.

Finally, the **FVEC approach** is to store the results according to both frequent-variable scheme and express-channel scheme. The cached data according to FVEC approach in Example 1 are listed in Table 1. The items cached due to the express-channel scheme are marked as red color.

4 Processing Inference Queries with Cached Results

We are now ready to illustrate the process of evaluating probabilistic inference queries with cached results.

4.1 Querying Algorithm

As explained in Sec. 3, the frequent results are cached in some nodes of INDSEP. The original algorithm of processing inference queries over INDSEP, as proposed in [8], needs to be revised in order to make use of cached results. Algorithm 1 lists the main steps of the revised querying algorithm, called *queryWithCache*. The revised parts are highlighted in red color.

```
1  if $v$ is a leaf node then
2      return extract($v$.potential,$q$);
3  if config.option = 'FVEC' then
4      $v' \leftarrow$ findExpressChannel($v$,$q$);
5      if $v'$ is NOT null then
6          return queryByChannel($v$,$v'$,$q$);
7  construct a steiner tree $T$ for $q$ over the children nodes of $v$;
8  $\Phi \leftarrow \emptyset$;
9  foreach node $v'$ in $T$ do
10     $q' \leftarrow S_{v'} \cup$ (all variables inside $v' \cap q$) ;
11     if $q'$ is a sub set of any shortcut $sc$ of $v$ or $v'$ then
12         $\Phi \leftarrow \Phi \cup$ extract($sc$,$q'$);
13         continue;
14     $r' \leftarrow$ searchInCache($q'$,$v'$);
15     if $r'$ is null then
16         $r' \leftarrow$ queryWithCache($v'$,$q'$);
17     $\Phi \leftarrow \Phi \cup \{r'\}$ ;
18 $r \leftarrow \frac{\prod_{\phi \in \Phi} \phi}{\prod_{s \in T} P(s)}$;
19 return $r$;
```

Algorithm 1. queryWithCache(Node v, Query q)

This algorithm works in a recursive manner. When a query q is evaluated, we invoke *queryWithCache* and set the root node as its input. Steps 1 and 2 are to check whether the input node v is on the leaf level. If the answer is YES, we can directly obtain the result by marginalizing q from v's potential. The function $extract(Factor, Query)$ is to extract the marginal distribution of a set of variables from a factor[1]. Steps 4 to 6 are only executed if the programme works in the 'FVEC' mode, i.e. we can use the *Express Channel* to skip some parts of the evaluation path. The details of leveraging express channels will be discussed in Sec. 4.2. If a query cannot be accelerated by express channel, we then build a Steiner tree at step 7, a path containing all variables in q. Step 8 is to initialize

[1] In this paper, we will use the terms "factor" and "distribution" interchangeably and ignore their difference in the probabilistic graphical model literatures.

an empty set for storing results of sub-queries. Next we evaluate each sub-query for every node v' contained by the Steiner tree (steps 9-17). Firstly, we build a sub-query (step 10) which contains the variables in v''s separators, or in the intersection of the variables in v' and in q. Steps 11-13 try to find the result in any short-cut of v or v'. If not found in short-cut, we then check whether the result could be found in cache (steps 14-16). In step 17, we insert the result of this sub-query into Φ. Finally, after collecting the results of each sub-query, we obtain the result by Equation 1.

The $searchInCache(q, v')$ function (step 14) tries to find a match of q in node v'. Here q *matches* a cached factor means that q is a subset of the variables contained in this factor.

4.2 Leveraging Express Channels

The use of *express channels* to accelerate query evaluation consists of two major steps. Firstly, we need to find whether there exists a channel and, if there are multiple ones, choose the longest one. This task is solved by the $findExpressChannel(v, q)$ function. A channel is represented by a children node of v, which means the item $P(S_v, S'_v)$ is contained in the cache of v. Therefore, the path from v to v' could be replaced by this channel factor. After that, we invoke the function $queryByChannel(v, v', q)$ to obtain the result of q which would cut off the path from v to v'.

Algorithm 2 lists the process of finding express channels. At the first step, we retrieve the lower common ancestor (LCA) of all variables in q because a channel must be shared by all the queried variables. Hence each common ancestor of all queried variables is a candidate to construct a channel, with the LCA generating the longest one. Steps 3 is to check whether the path $< v, v' >$ exists, i.e. the joint distribution of $S_v \cup S'_v$ is cached. If not, we then set v' as its parent node and repeat this process. The loop ends if we find v' becomes v, which means we could not find a children node of v which can build a channel. A *null* value will be returned for this case.

The $queryByChannel(v, v', q)$ function is not complicated. As listed in Algorithm 3, the input query q is decomposed into two parts. The first part, i.e. (v', q) is obtained by invoking the $queryWithCache$ function, which is not costly since v' usually locates in the levels very close to the leaf node. The second part, i.e. (v, v'), can be loaded from cache. The final result is then obtained by combining these two parts with Equation 1 (step 3).

5 Experimental Study

We now report our experimental results. All programs are implemented in C++ under gcc 4.1.2, and run on a machine with six-core 2.4G Hz Intel(R) Xeon(R) CPU, 16G RAM and Linux 2.6.18. For comparison, we implement the INDSEP index [8] based on an open source C++ library libDAI[2]. Without otherwise specifications, each point in the following figures is an average of 200 runs.

[2] `http://cs.ru.nl/~jorism/libDAI/`

```
1 $v' \leftarrow getLCA(q)$ ;
2 while $v'$ != $v$ do
3     if $P(S_v \cup S'_v)$ can be found in cache then
4         return $v'$;
5     $v' \leftarrow$ the parent node of $v'$;
6 return null;
```

Algorithm 2. findExpressChannel(Node v, Query q)

```
1 $r_1 \leftarrow queryWithCache(v', q \cup S_{v'})$ ;
2 $r_2 \leftarrow searchInCache(S_v \cup S_{v'}, v)$ ;
3 $r \leftarrow \frac{r_1 r_2}{\prod_{s \in S_{v'}} P(s)}$ ;
4 return $r$;
```

Algorithm 3. queryByChannel (Node v, Node v',Query q)

Datasets and Queries. We generate a probabilistic database with one relation and 100,000 tuples, which is represented in GRN model. The uncertain data is generated in the way that in GRN, each generator or recognizer is linked to 2-4 tuple variables. For comparison, we also generate probabilistic relations with 10K, 20K, and 50K tuples respectively[3]. The inference queries are all about extracting the joint distribution of a set of random variables. The number of variables contained in each query varies from six to ten, and on average a query covers more than 60% range of the whole INDSEP. The queries are generated by following the *Pareto* pattern, which means that among all queries 80% queried variables come from a set containing 20% of the whole variables.

5.1 Parameter Tuning

Firstly, we illustrate our results for choosing appropriate node size, i.e. the maximum number of cliques contained in each INDSEP node. Note that our proposed approach is main-memory based. It seems as if the larger node size, the shorter INDSEP tree, and the better query performance.We test different node sizes on querying INDSEP on the 100K dataset. As illustrated by Figure 3, the performance does not keep increasing as the node size enlarges. The reason is due to the high computation cost of handling node potentials during query evaluation. This cost will increase exponentially when the node size enlarges and may totally counteract the benefits obtained from the reduction of tree height. Based on this observation, we set the node size as eight, the optimal point in Figure 3.

[3] By default, the dataset contains 100K tuples.

We further run a series of experiments to detect the optimal value of cache size, i.e. the maximum memory consumption permitted to store all the cached factors. From Fig. 4, we can see that when the cache size increases, the response time of both *SubQuery* and *FVEC* will drop in the initial phase, then increase a little bit and finally become stable. When cache size is small, a larger cache will accommodate more factors and help reduce the response time. But if the cache size is big enough to store most frequent results, the performance cannot be increased more. On the contrast, the response time may be longer since there are many infrequent results reserved in cache and more efforts are needed to search the cache. As implied by the results, we set the cache size as 200M for the following experiments. We also compare the cache hit ratio, i.e. percent of intermediate sub-queries which are skipped due to matched factors in the cache. We can observe from Fig.5 that the cache hit ratio of *FVEC* is much higher than that of *SubQuery* which is because the advantages of using smaller factors and express channels. Also, the ratios will be become stable for both methods when the cache size is big enough, which again explains why the performance will not increase constantly when enlarging cache sizes.

5.2 Performance Comparison

Overall Performance. Figure 6 shows the comparison of three approaches, i.e. INDSEP, *Subquery* and *FVEC* on the 100K data set. The x-axis shows the sequence number of queries. Basically a sequence is a group of 200 queries. The y-axis shows the average response time for each sequence. Clearly, the *FVEC* outperforms the other two approaches. The relative improvement with respect to INDSEP is more than 50 percents. The *Subquery* approach performs worse than *FVEC* but better than INDSEP. This result is coincident with our estimation. Our proposed approach will increase the cache hit ratio as discussed in Section 4 and it wins the best. The *Subquery* approach is based on the INDSEP and tries to utilize cached results for answering queries, so it defeats INDSEP.

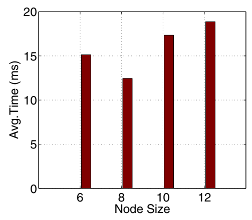

Fig. 3. Time vs. NodeSize

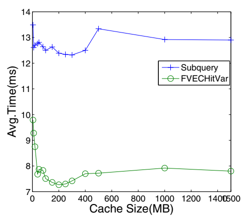

Fig. 4. Time vs. Cache

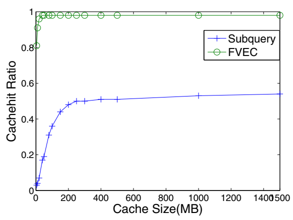

Fig. 5. HitRatio vs. Cache

In order to illustrate the fire up phase of collecting frequent variables, we partially enlarge Figure 6 and show performance of processing the first 1,000 queries in Figure 7. Each point of Figure 7 is an average of 20 runs. From this figure, we can see that in the beginning, the performance of the three approaches are roughly the same. During the processing of queries, the response time of *Subquery* and *FVEC* will decrease quickly, while the performance of INDSEP has no obvious changes. This because both *Subquery* and *FVEC* will cache some results and use them for answering new queries. Hence their performance will become better when experiencing more queries.

Finally, we test the average response time of the three approaches on four datasets for processing 10,000 queries, with sizes of 10K, 20K, 50K and 100K respectively. Figure 8 illustrates the result. We can find that *FVEC* always performs the best among the three approaches, and the increasing rate is acceptable.

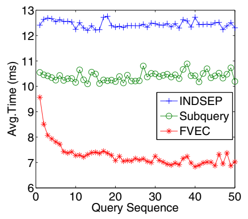

Fig. 6. Comparison of Overall Performance

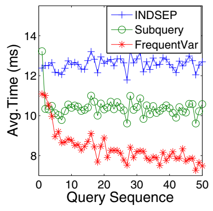

Fig. 7. Fire Up Phase of Figure 6

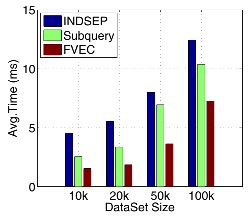

Fig. 8. Response Time vs. Data Set Size

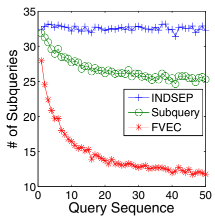

Fig. 9. # of Subqueries

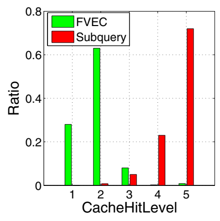

Fig. 10. Cache Hit Levels

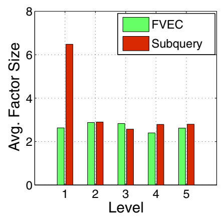

Fig. 11. Avg. Factor Size

Breakdown Analysis. We now illustrate why *FVEC* could perform the best through several breakdown analysis. Firstly, we want to mention that during a query evaluation the number of extended subqueries is one of the most crucial factors which affect the response time. We compare the number of required subqueries for the three approaches in Figure 9. Again *FVEC* performs the best. Recall that *FVEC* can reduce the number of subqueries by completely matching subqueries with cached result, and can skip many subqueries by the express channels based on Algorithm 3.

As discussed before, a subquery may be again broken into a set of children subqueries if it is issued on an non-leaf node. Hence on average a cache hitting in an upper level could save much more subqueries than hittings in lower levels. To illustrate this, we further compare the cache hit levels of the two caching approaches in Figure 10. Here a cache hit happens if and only if a subquery terminates by reusing some cached information. The y-axis of Figure 10 presents the percents of cache hits at each level. From Figure 10, we can see that more than 90% cache hittings of *FVEC* happens in the first two levels, while most cache hittings of *Subquery* appear in the fourth and fifth levels. That explains why *FVEC* performs much better than *Subquery*.

Finally, we compare the average size of factors, i.e. the number of variables in a factor. Note that the *FVEC* approach utilizes smaller factors to accommodate more cached items 3.2. This motivation is verified by the results in Figure 11. At the first level, the average factor size of *Subquery* is much larger than that of *FVEC*. We also find that this effect is not that evident in lower levels. Note that a sub-query evaluated in the first level would be decomposed into several smaller sub-queries to the second level. Hence the size of sub-queries in lower levels would be quite small (usually one or two) and this is why it is hard to see the difference in these levels. Remember that the first level would be the most important place to kill sub-queries. Thus *FVEC* could benefit a lot from the larger capacity in the first level.

6 Related Works

Management of Uncertain Data. During the past decade, along with the quick development of applications like mobile data management, sensor network, data streams and data integration, processing uncertain data attract many research interests from the database community. These works are mainly targeted on uncertain data models [2], probabilistic queries [4,11], uncertain data index [17,8], and uncertain data mining [16].

Querying Uncertain Data. Evaluating probabilistic queries is known to be #-complete [5]. Many solutions are proposed for improving the performance, including Monte-Carlo sampling [4], verification and refinement approaches [19], and uncertain data index [17] etc.. All of these works assume that there are no correlations or only locally correlations, i.e. each uncertain data is only correlated with several data items, and independent of others. Hence they are not applicable for the applications where uncertain data are highly correlated with each other.

Works on querying correlated uncertain data usually represent uncertain data by probabilistic graphical models (PGM) and regard this problem as performing inference on PGM. Sen et al. utilize the shared correlations among uncertain data to simplify the original PGM and reduce the query evaluation time [13]. In [8], the INDSEP structure is proposed for indexing correlated uncertain data, which is adopted by this work. In [12], probabilistic inference is transformed to SAT and existing solutions for SAT could be adopted for solving inference. None of these works consider the use of cached results for accelerating future query evaluation. The work most related to ours is [15]. Instead of storing intermediate results as [15] does, we cache the results of each frequent variable in order to increase the cache hit ratio. Also, we try to cut off unnecessary path from the evaluation path with the *express channel* technique.

7 Conclusion and Future Works

In this paper, we address the problem of answering probabilistic inference queries over large volume of uncertain data with high correlations. In order to solve this problem, we propose to cache the result of each frequent single variable in

INDSEP and design a series of novel techniques for efficient processing inference queries with cached information. According to our experimental results, our proposed approach is both effectiveness and efficient compared with the state-of-the-art works. About the future work, we plan to study how to further improve query performance with some approximates and/ or cloud computing techniques.

Acknowledgments. This work is funded by the National Science Foundation of China under Grant No. 61003086, and National Basic Research Program of China (973 Program) No. 2012CB316205.

References

1. The pareto principle, `http://en.wikipedia.org/wiki/Pareto_principle`
2. Chen, R., Mao, Y., Kiringa, I.: Grn model of probabilistic databases: construction, transition and querying. In: SIGMOD 2010, pp. 291–302 (2010)
3. Cheng, R., Chen, J., Mokbel, M.F., Chow, C.-Y.: Probabilistic verifiers: Evaluating constrained nearest-neighbor queries over uncertain data. In: ICDE 2008, pp. 973–982 (2008)
4. Cheng, R., Kalashnikov, D.V., Prabhakar, S.: Evaluating probabilistic queries over imprecise data. In: SIGMOD 2003, pp. 551–562 (2003)
5. Dalvi, N.N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: VLDB 2004, pp. 864–875 (2004)
6. Das Sarma, A., Dong, X., Halevy, A.: Bootstrapping pay-as-you-go data integration systems. In: SIGMOD 2008, pp. 861–874 (2008)
7. Hwang, F.K., Richards, D.S.: Steiner tree problems. Networks 22(1), 55–89 (1992)
8. Kanagal, B., Deshpande, A.: Indexing correlated probabilistic databases. In: SIGMOD 2009, pp. 455–468 (2009)
9. Kanagal, B., Deshpande, A.: Lineage processing over correlated probabilistic databases. In: SIGMOD 2010, pp. 675–686 (2010)
10. Koller, D., Friedman, N.: Probabilistic Graphical Models Principles and Techniques. MIT Press, London (2009)
11. Lian, X., Chen, L.: Efficient query answering in probabilistic rdf graphs. In: SIGMOD 2011, pp. 157–168 (2011)
12. Sang, T., Bearne, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: AAAI 2005, pp. 475–481. AAAI Press (2005)
13. Sen, P., Deshpande, A., Getoor, L.: Exploiting shared correlations in probabilistic databases. In: VLDB 2008 (2008)
14. Singh, S., Mayfield, C., Prabhakar, S., Shah, R., Hambrusch, S.E.: Indexing uncertain categorical data. In: ICDE 2007, pp. 616–625 (2007)
15. Song, S., Chen, L., Yu, J.X.: Answering frequent probabilistic inference queries in databases. IEEE Trans. on Knowledge and Data Engineering 23, 512–526 (2011)
16. Sun, L., Cheng, R., Cheung, D.W., Cheng, J.: Mining uncertain data with probabilistic guarantees. In: KDD 2010, pp. 273–282 (2010)
17. Tao, Y., Cheng, R., Xiao, X., Ngai, W.K., Kao, B., Prabhakar, S.: Indexing multi-dimensional uncertain data with arbitrary probability density functions. In: VLDB 2005, pp. 922–933. VLDB Endowment (2005)
18. Wang, D.Z., Franklin, M.J., Garofalakis, M., Hellerstein, J.M.: Querying probabilistic information extraction. In: PVLDB, vol. 3(1-2), pp. 1057–1067 (September 2010)
19. Zhang, W., Lin, X., Zhang, Y., Pei, J., Wang, W.: Threshold-based probabilistic top-k dominating queries. The VLDB Journal 19(2), 283–305 (2010)