

Sophisticated Indexes for Implementing Probabilistic Expert Systems

C.J. Butz and F. Fang

Department of Computer Science, University of Regina
Regina, Saskatchewan, Canada S4S 0A2
{butz, fang11fa}@cs.uregina.ca

Abstract. Indexes are crucial for the efficient implementation of probabilistic expert systems. However, the indexes previously proposed, and the methods for applying them, are somewhat elementary. Moreover, recent experiments involving large Bayesian networks have resulted in the computer running out of memory. This further emphasizes the importance of indexes for probabilistic inference involving secondary memory. In this paper, we propose several sophisticated index structures for implementing probabilistic expert systems. We show the advantages of these advanced indexes over the previously proposed indexes. In addition, by discussing the advantages and disadvantages of each of our sophisticated index structures, the designer of a probabilistic expert system can choose the index structure most appropriate for the particular probabilistic network under consideration.

1 Introduction

Bayesian networks [6] are an established framework for uncertainty management in artificial intelligence. A Bayesian network consists of a *directed acyclic graph (DAG)* and a corresponding set of *conditional probability tables (CPTs)*. The *probabilistic conditional independencies* [9] encoded in the DAG indicate that the product of the CPTs is a *joint probability distribution*.

Huang and Darwiche [3] explicitly state that the use of indexes is *crucial* for an efficient implementation of a probabilistic expert system, which manages the physical probability distributions in accordance to the schema of the given Bayesian network (or Markov network [8]). In particular, the key task of multiplying two distributions in the Hugin architecture [4] is facilitated by the introduction of two index structures, called *cluster-sepset* and *shrink mapping*. While the cluster-sepset index assists join tree probability propagation in the Hugin architecture, we show that it is not beneficial for answering some queries after propagation finishes. Also, the shrink mapping for handling evidence is very trivial. The suggestion of creating an shrink index for each piece of collected evidence is questionable. Moreover, in the experimental studies conducted by Zhang and Poole [11], it was reported that the computer ran out of memory when using the Hugin method on large Bayesian networks. The above discussion indicates a clear need for advanced indexes when implementing probabilistic expert systems.

In our previous research [2, 8], we have argued that the implementation of probabilistic expert systems can take full advantage of the corresponding work done in implementing *relational database management systems* (RDMSs). The soundness of this argument is based on [9], in which it was shown that the logical implication of probabilistic conditional independence coincides exactly with that of embedded multivalued dependency in the classes of Bayesian networks and Markov networks. Since the tuples actually joined when multiplying two probability distributions are precisely defined as the natural join of the two distributions [8], it is perhaps worthwhile to adopt the many indexes, developed in RDMSs for implementing natural join, for implementing multiplication in probabilistic expert systems.

In this paper, we begin by considering a naive implementation of multiplication. This naive approach can be improved by first sorting the tuples in the two probabilistic relations. An even more efficient implementation of multiplication is shown by incorporating an *index*. More importantly, we then propose sophisticated implementations of multiplication. In particular, we examine four kinds of advanced indexes, namely, *dense*, *sparse*, *multilevel sparse*, and *secondary*. The dense index is used to facilitate multiplication, and in that sense is loosely related to the cluster-set index. Huang and Darwiche [3] construct a new shrink mapping for each piece of collected evidence and use it to delete those rows disagreeing with the evidence. On the other hand, one sparse index is useful as it imposes little maintenance overhead for the deletion and insertion of tuples online. To facilitate the efficient retrieval of tuples from secondary memory, we propose using multilevel sparse indexes. This type of index is especially useful for large Bayesian networks that cannot reside in main memory, such as those reported in [11]. Finally, we advocate secondary indexes for the efficient processing of common queries. By elaborating on the strengths and weaknesses of these suggested indexes, the designer of a probabilistic expert system can choose the most appropriate index for the current probabilistic network under consideration. The work here further demonstrates the cross-harmonization of the Bayesian network and relational databases communities [2, 8, 9, 10].

This paper is organized as follows. Section 2 reviews background information. Naive implementations of multiplication are presented in Section 3. In Section 4, we suggest more sophisticated approaches to implementing the multiplication operator in probabilistic expert systems. In Section 5, we give thorough discussion on the advantages and disadvantages of these indexes, compared both with each other and the previously proposed indexed. The conclusion is presented in Section 6.

2 Background Knowledge

In this section, we review the notions of two index structures for implementation in probabilistic expert systems.

Probabilistic inference, however, is usually carried out in a join tree constructed from a given Bayesian network. A *join tree* [5] is a tree with the property that any variable in two nodes is also in every *separating set* on the path between the two.

Huang and Darwiche [3] explicitly state two indexes, called *cluster-sepset* and *shrink mapping*, are central to an efficient implementation of probabilistic expert systems. Probabilistic inference involves two kinds of operations, namely, addition and multiplication. The particular configurations to be summed and multiplied depend, in part, on the structure of the join tree.

The key to implementing join tree probability propagation is to locate the corresponding configurations of the potentials for a join tree node [3] and join tree separator. Given potentials $\Phi(X)$ and $\Phi(S)$ on a join tree node X and join tree separator S , a *cluster-sepset* mapping $\mu_{x,s}$ is a mapping from the configurations x of $\Phi(X)$ to the corresponding configurations S of $\Phi(S)$.

Example 1: Consider the join tree node potential $\Phi(h, k, l, r)$ and join tree separator potential $\Phi(h, l, r)$ in Fig. 1. The cluster-sepset index $\mu_{x,s}$ maps configurations of $\Phi(h, k, l, r)$ to corresponding configurations of $\Phi(h, l, r)$.

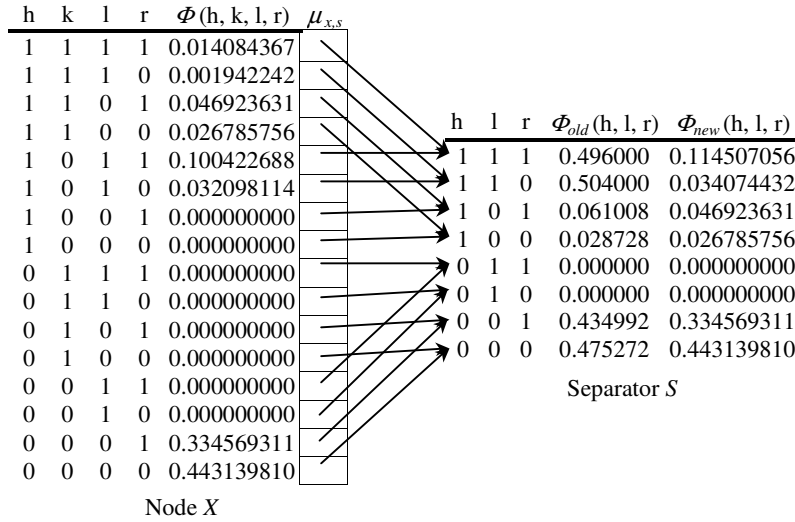


Fig. 1. Cluster-sepset mapping [3] $\mu_{x,s}$ for node $X = hklr$ and separator $S = hlr$

The next example demonstrates the usefulness of the cluster-sepset index.

Example 2: Suppose potential $\Phi(X)$ and $\Phi(S)$ in Fig. 1 need be multiplied. The cluster-sepset index removes the requirement of determining which configurations of $\Phi(X)$ and $\Phi(S)$ are to be multiplied.

The cluster-sepset index, however, is not useful for answering some queries, as the following example clearly shows.

Example 3: Suppose query $p(h, r)$ is issued in our running example. The cluster-sepset index between $\Phi(h, k, l, r)$ and $\Phi(h, l, r)$ is irrelevant to the processing of $p(h, r)$.

Suppose we observe evidence $E = e$. The configurations disagreeing with $E = e$ are deleted, that is, the probabilities are set to zero. Without any additional modifications in implementation, the tuples with zero probability will continue to be visited, yet will have absolutely no impact on the probabilities computed. Thus, we seek further changes in implementation to avoid accessing tuples with zero probability.

The change in implementation to handle evidence is very trivial. A second index, called a *shrink mapping*, is used to point to the tuples with positive probability. The effect is that the size of the probability table at the node is shrunk, thereby improving the running time of any message passing through this join tree node.

Example 4: In our running example, suppose evidence $r = 1$ is collected. A second index σ_x is created for the potential $\Phi(X)$ at join tree node $X = hklr$.

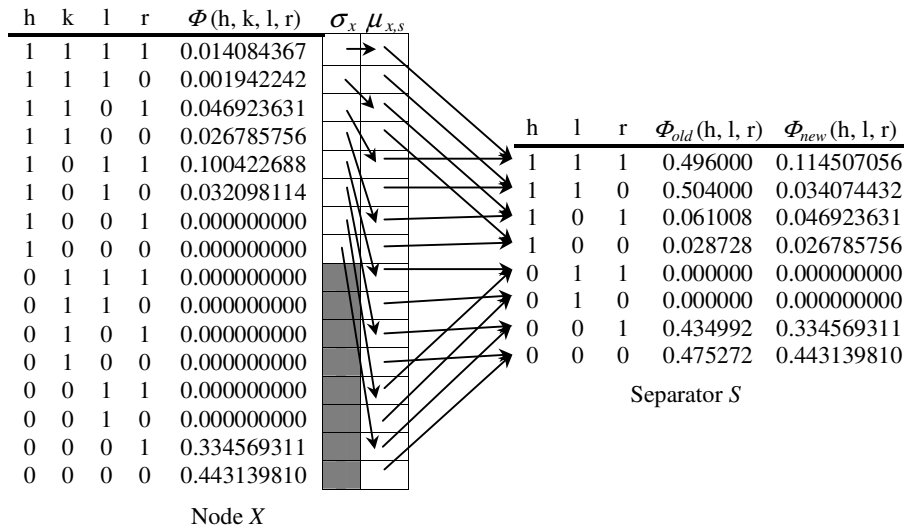


Fig. 2. Shrink mapping [3] on evidence $r = 1$ for join tree node $X = hklr$

More recently, we have shown [2] how the select operator in RDMSs can be implemented in probabilistic expert systems to lower the number of multiplications needed for handling evidence, a common goal in query optimization [5]. Even when multiplication is required, we can still draw from the experience learned in RDBMs.

Note that the tuples actually multiplied are precisely the joinable tuples. Let $r_1 (R_1)$ and $r_2 (R_2)$ be two relations. Let $R = R_1 R_2$ and $X = R_1 \cap R_2$. We say tuples $t_1 \in r_1$ and $t_2 \in r_2$ are *joinable* on X , if there is a tuple t on X such that $t_1 = t(R_1)$ and $t_2 = t(R_2)$. (Hence, the joinable notion formalizes the term *corresponding* in [3].) For this reason, we seek to exploit the techniques available in RDMSs for efficiently implementing natural join to efficiently implement multiplication in probabilistic expert systems.

3 Naive Implementations of Multiplication

We gradually taper from naive implementations of multiplication to more efficient ones. The discussion here draws from [1].

For simplified discussion, we will henceforth focus on computing the product join of the relations for two neighbouring join tree nodes and ignore the probably table of the separator between the two nodes.

The most naive approach to computing $r = r_1(X) \otimes r_2(Y)$ is to compare each row of $r_1(X)$ with each row of $r_2(Y)$. For instance,

```
r := ∅;
for each u in r1(X)
  for each v in r2(Y)
    if u and v are joinable
      r = r ∪ {u ⊗ v}
```

Implementing multiplication in this fashion will take in the order of the product $n_1 \times n_2$ of the sizes of input relations $r_1(X)$ and $r_2(Y)$.

Example 5: Suppose we wish to compute the product join $r = r(h, k, l, r) \otimes r(b, h, l, r)$ for join tree nodes $hklr$ and $bhlr$. Following the naive approach, we have:

```
r := ∅;
for each u in r1(h, k, l, r)
  for each v in r2(b, h, l, r)
    if u and v are joinable
      r = r ∪ {u ⊗ v}
```

An obvious suggestion is to first sort the two distributions before computing the product. The *sort-merge* algorithm, which independently sorts both inputs according to the join attributes and then performs a simultaneous scan of both relations, can be used to output joinable tuples as discovered. This reduces the running time to the order of $\max(n_1 \log n_1 + n_2 \log n_2, \text{size of output})$.

Indexes, however, provide an even more efficient implementation of multiplication. In the above naive approach, replace the inner loop by indexed retrievals to tuples of $r_2(Y)$ that are joinable with the tuple of $r_1(X)$ under consideration. Assuming that a small number of tuples of $r_2(Y)$ match a given tuple of $r_1(X)$, this computes the join in time proportional to the size of $r_1(X)$.

4 Sophisticated Implementation of Multiplications

In this section, we examine implementing multiplication in probabilistic expert systems using four kinds of sophisticated indexes, namely, *dense*, *sparse*, *multilevel* and *secondary*. Our discussion on implementing multiplication closely follows the corresponding discussion on implementing natural join [1, 7].

4.1 Dense Index Structures

Given that a file is sorted in sequential order, a *primary* index is one whose search-key also defines the sequential order of the file. An *index order* consists of a search-key value, and pointers to one or more records with the same search-key value. Primary indexes can be classified as either *dense* or *sparse* (discussed in Section 4.2).

A dense index has a record for each search-key value in the file. More specifically, an index record consists of a search-key value and a pointer to the first data record with that search-key value. Any remaining data records with the same search-key are stored sequentially after the first data record.

We propose implementing multiplication using a dense index as follows. Recall the naive implementation of multiplication. The distribution corresponding to the outer loop can be sequentially ordered, though this is not necessary. The search-key is the set of variables in the separator between the neighbouring join tree nodes. Sort the second distribution, the distribution corresponding to the inner loop, according to the values of the search-key. Assign pointers in this distribution from each data record to the next data record until the end-of-file. Create a dense index for the variables in the separator. Sort the values in the index in the same order as done for the second distribution. For each index record, assign the pointer to the first data record in the second distribution with the same search-key value.

Example 6: Suppose we want to construct a dense index for computing the product $p(h, k, l, r) \cdot p(h, l, r, b)$. In the naive implementation of multiplication, let $p(h, k, l, r)$ be the distribution corresponding to the outer loop. Similarly, let $p(h, l, r, b)$ be the distribution corresponding to the inner loop. The remainder of this example is illustrated in Fig. 3. As the separator is *hlr*, sort $p(h, l, r, b)$ accordingly. Assign pointers in $p(h, l, r, b)$ to the next data record until the end-of-file. A dense index is created on variables *hlr*. There is a search-key value in the dense index for each value of *hlr* in $p(h, l, r, b)$. In the dense index, assign the pointer for each search-key value to the first record in $p(h, l, r, b)$ with the same *hlr* value.

After the dense index has been constructed, perform the following when the two distributions need be multiplied. For each row in the first distribution (corresponding to the outer loop), locate the corresponding search-key value in the dense index. Multiply the row under consideration in the first distribution with the row pointed to by the search-key value in the dense index. Multiply the row under consideration in the first distribution with each subsequent record in the second distribution until a different search-key value is reached.

Example 7: Let us use the dense index in Fig. 3 to compute the product $p(h, k, l, r) \cdot p(h, l, r, b)$. Consider the first row of $p(h, k, l, r)$, say $\langle h: 1, k: 1, l: 1, r: 1, 0.014084367 \rangle$. The corresponding search-key value in the dense index is $\langle h: 1, l: 1, r: 1 \rangle$. The first data record pointed to by this index record is $\langle h: 1, l: 1, r: 1, b: 1, 0.08501664 \rangle$. These two data records are multiplied giving $\langle h: 1, k: 1, l: 1, r: 1, b: 1, 0.0011974055886688 \rangle$. The next data record in $p(h, l, r, b)$ is $\langle h: 1, l: 1, r: 1, b: 1, 0.028905392 \rangle$. This data record and the first data record in $p(h, k, l, r)$ are similarly

multiplied. As the next subsequent data record in $p(h, l, r, b)$, i.e., $\langle h: 1, l: 1, r: 0, b: 1 \rangle$, has a different search-key value $\langle h: 1, l: 1, r: 0 \rangle$, we move to the second row in the first distribution $p(h, k, l, r)$, and so on.

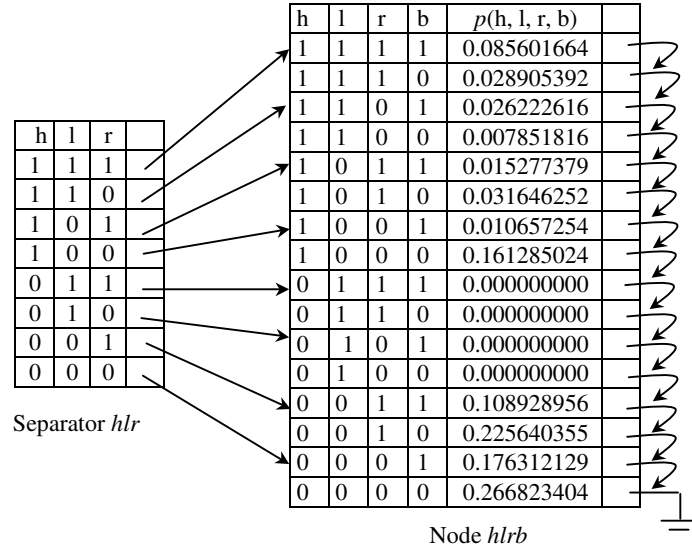


Fig. 3. A dense index into $p(h, l, r, b)$ in order to implement the multiplication of $p(h, k, l, r)$ and $p(h, l, r, b)$ for join tree nodes $hklr$ and $hlrb$

4.2 Sparse Index Structures

A sparse index is a slight variation of the dense index structure presented in the last subsection.

Recall that a dense index has a search-key value for each value of the search-key appearing the distribution. Clearly, the dense index can grow large whenever the distribution being indexed is large. For instance, the large Bayesian networks discussed in [11] resulted in a system crash, as the computer ran out of memory. As the management of a dense index can become unwieldy for large Bayesian networks, we introduce the notion of sparse index.

A *sparse* index is a dense index except that an index record appears only for *some* search-key values in the distribution. A discussion on the construction of a sparse index is omitted, as it is very similar to that of a dense index. An example of a sparse index into $p(h, l, r, b)$ is given in Fig. 4.

We suggest utilizing a sparse index to implement multiplication as follows. We assume that the second distribution, corresponding to the inner loop of the naive implementation, is sorted in descending ordering of the search-key values. For each row in the first distribution, corresponding to the outer loop in the naive implementation, find the index record with the smallest search-key value that is greater than or equal to the search-key value being sought. Start at the data record in

the second distribution being pointed to by index record, and follow the distribution pointers until the desired search-key value is found. Multiplication then proceeds as with a dense index.

Example 8: Let us use the sparse index in Fig. 4 to compute the product $p(h, k, l, r) \cdot p(h, l, r, b)$ for the neighbouring nodes $hklr$ and $bhlr$. For pedagogical reasons, consider row $\langle h: 1, k: 1, l: 0, r: 0, 0.026785756 \rangle$ in $p(h, k, l, r)$. The search-key $\langle h: 1, l: 0, r: 0 \rangle$ does *not* appear in the sparse index. Instead, the smallest search-key value that is greater than $\langle h: 1, l: 0, r: 0 \rangle$ appearing in the sparse index is $\langle h: 1, l: 0, r: 1 \rangle$. We follow the pointer for this index record to the data record $\langle h: 1, l: 0, r: 1, b: 1, 0.015277379 \rangle$. The distribution pointers are followed until we reach the proper search-key value in data record, that is, $\langle h: 1, l: 0, r: 0, b: 1 \rangle$ in $p(h, l, r, b)$. Multiplication is carried out from this point in the same manner as for a dense index.

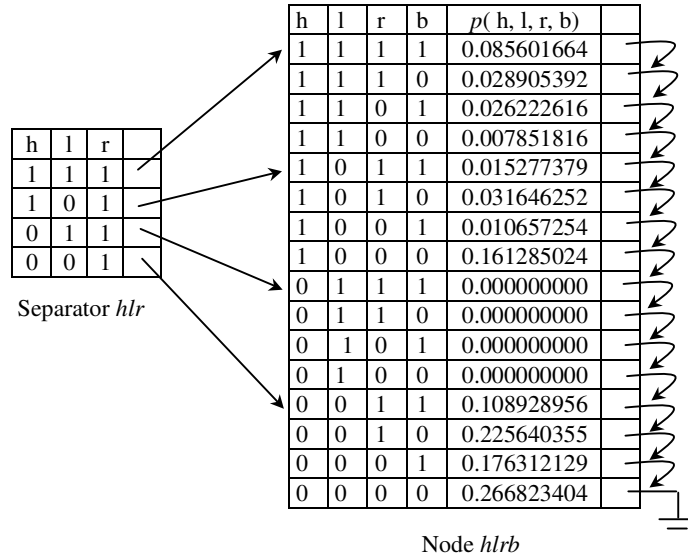


Fig. 4. A sparse index into $p(h, l, r, b)$ in order to implement the multiplication of $p(h, k, l, r)$ and $p(h, l, r, b)$

4.3 Multilevel Index Structures

In this section, we present a third index, called *multilevel*, for implementing multiplication. In contrast to the discussion on typical implementation [3, 8], this is the first discussion in the probabilistic reasoning literature tackling the problem of secondary memory management.

The search time is costly for a sparse index that need be stored in secondary memory due to its large size [7]. A typical search may involve several disk block reads. If *overflow blocks* [7] have not been used, then a binary search can be conducted, but still with a large cost. Otherwise, a sequential search can be

undertaken but this normally takes even longer. The solution to this problem is to create indexes into indexes themselves.

A *multilevel index* is an index with more than one level. For simplicity, we will concentrate on a two-level index. An example of a two-level index is shown in Fig. 5.

We suggest the following approach to applying multilevel indexes for the efficient implementation of multiplication in probabilistic expert systems. As usual, sort the second distribution according to the search key value. Following this order, assign data records of the distribution to blocks of memory such that all records with the same search-key appear together in the same block. Unlike traditional sparse indexes, the sparse indexes used in a multilevel index are stored contiguously in memory. This removes the necessity of pointers from one index record to the next in each index. The search process is similar to that for a single sparse index except for the multiple levels.

Example 9: In our usual example, suppose we want to implement the product $p(h, k, l, r) \cdot p(h, l, r, b)$ using a two-level sparse index into $p(h, l, r, b)$. As depicted in Fig. 5, let the outer index be on variable h , while the inner index is on variables hlr .

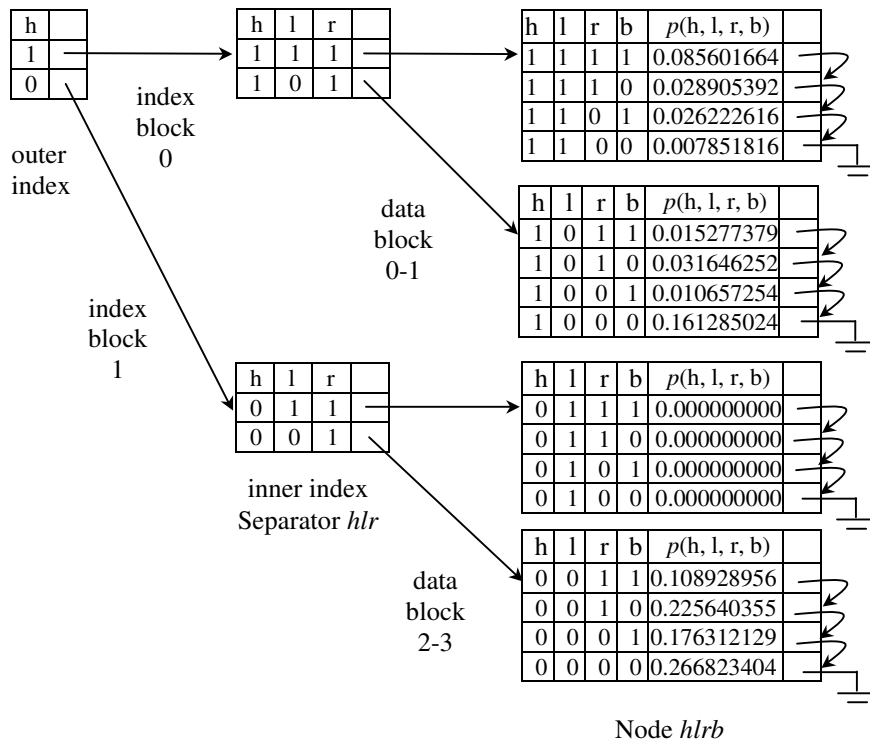


Fig. 5. A two-level sparse index into $p(h, l, r, b)$ in order to implement the multiplication of $p(h, k, l, r)$ and $p(h, l, r, b)$

4.4 Secondary Index Structures

Whereas the indexes previously proposed emphasize the efficient multiplication of the probability tables for two neighbouring join tree nodes, in this section we present the *secondary index* structure to facilitate the efficient processing of common queries.

Some variables may be of more interest than other variables in a probabilistic network. For instance, in a large Bayesian network for medical diagnosis, one may be interested in constantly monitoring the specific variable *blood-pressure*, as evidence is continually collected. Thereby, it is beneficial to construct an index to help process queries such as $p(\text{blood-pressure} = \text{high} \mid E=e)$, where $E = e$ is the evidence collected at the time the query is posed.

A secondary index is an index satisfying the following two properties. First, there is an index value for each search-key value in the distribution. Second, there must be a pointer to every record in the distribution. We can use an extra level of indirection to implement secondary indexes.

Example 10: In our running example, suppose variable r is *blood-pressure* and that we seek to constantly monitor r as evidence $E = e$ is collected. To help process the query $p(r \mid E=e)$ efficiently, we can utilize a secondary index on r , as shown in Fig. 6.

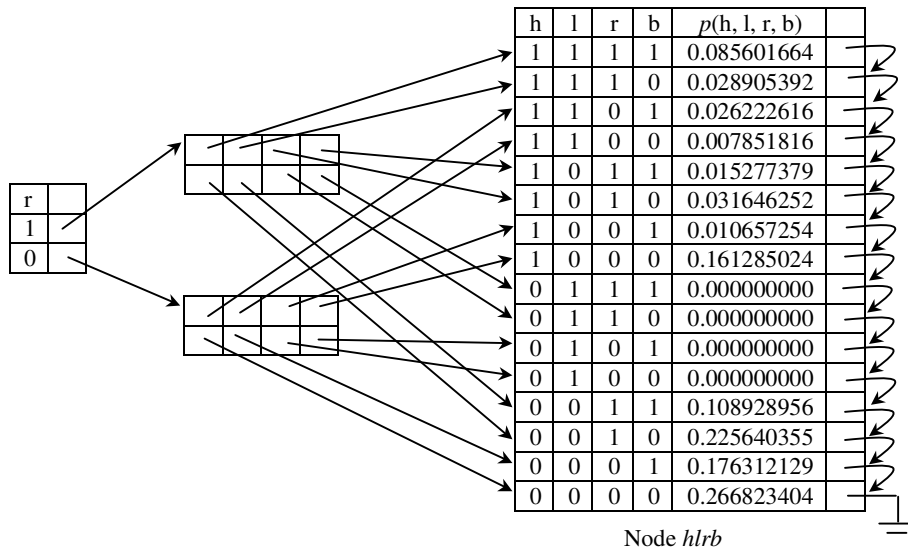


Fig. 6. Using a secondary index to efficiently answer the common query $p(r \mid E=e)$, where $E = e$ is the collected evidence

Example 10 clearly demonstrates that secondary indexes are useful in practice when one is interested in constantly monitoring certain variables in a probabilistic network.

5 Advantages and Disadvantages of the Index Structures

Indexes are central to the efficient implementation of a probabilistic expert system [3]. In this section, we examine the advantages and disadvantages of those indexes proposed in [3] and in this manuscript.

The cluster-sepset index given in [3] is beneficial for implementing join tree probability propagation using the Hugin architecture [4]. It is not clear, however, how this index is useful for more recent architectures for join tree probability propagation such as the Lazy architecture [5]. That is, the cluster-sepset mapping is tailored for one distribution per separator, while each separator may have multiple distributions in the Lazy approach. Moreover, Example 3 explicitly demonstrated that the cluster-sepset index is not beneficial for processing queries in general.

Huang and Darwiche [3] proposed the shrink mapping index to handle collected evidence. It is suggested that a shrink mapping index be constructed for each piece of collected observations. That is, create n indexes, if the values of n variables are observed. Such an approach may not be feasible, as n grows large. Instead, it is perhaps better to create a single index and utilize the numerous techniques developed for the insertion and deletion of tuples online [7].

Regarding the use of dense and sparse indexes suggested in this work, it is generally faster to locate a record using a dense index as compared to a sparse index. However, a sparse index has an advantage over a dense index in that they require less space and impose less maintenance overhead for insertions and deletions [7].

The unquestionable necessity of multilevel sparse indexes for secondary memory management was shown by Zhang and Pool [11]. In that study, it was reported that the computer ran out of memory when conducting experiments on join tree probability propagation for large Bayesian networks. As these large Bayesian networks cannot reside in main memory, they must be stored in secondary memory. As record retrieval is quite costly for large sparse indexes, due primarily to the number of blocks needed to be read, multilevel sparse indexes are utilized for faster retrieval. Since this is the first work to mention secondary memory management in the implementation of probabilistic expert systems, there is no other index to be compared with. It is known in the database community, however, that a multilevel index is advantageous as it requires significantly fewer I/O operations than does searching for records by binary search of the table [7].

The salient feature of a secondary index is that it improves the performance of queries involving search keys other than the primary one. The disadvantage of implementing a secondary index is an increase in space usage. Nevertheless, we have shown in Example 10 that secondary indexes have legitimate usefulness in probabilistic expert systems.

6 Conclusion

Although Bayesian networks [6] have been applied for a variety of problems in artificial intelligence, there is little literature on the implementation of probabilistic expert systems [3, 8]. Huang and Darwiche [3] explicitly state that indexes are critical

for the efficient implementation of probabilistic expert systems. As the cluster-sepset index [3] seems to be tailored for the Hugin architecture (see Section 5), it is not clear how it can be utilized in recent architectures such as the Lazy architecture [5]. In addition, we have shown in Example 3 that the cluster-sepset index is not useful for processing some queries after join tree propagation terminates. Moreover, the manageability of building a shrink mapping [3] for each piece of collected evidence is debatable given a large number of observed variables. An alternative approach is to build one index and then take advantage of the many techniques developed for the insertion and deletion of tuples [7].

We have presented four sophisticated indexes for implementing multiplication in probabilistic expert systems. The advantages and disadvantages of each index are discussed in Section 5. In particular, this is the *first* work to suggest multilevel indexes for those Bayesian networks that are too large to reside in main memory [11]. In addition, this is also the *first* work to propose creating secondary indexes for constantly monitoring the most important variables in a probabilistic network. The work here takes full advantage of the intrinsic relationship between Bayesian networks and relational databases [2, 8, 9, 10].

References

1. Abiteboul, S., Hull, R., and Vianu, V.: Foundations of Databases. Addison-Wesley, United States of America (1995)
2. Butz, C.J. and Fang, F.: Incorporating Evidence in Bayesian networks with the Select Operator. In Proceedings of the 18th Canadian Conference on Artificial Intelligence, Springer-Verlag, Victoria British-Columbia (2005) 297-301
3. Huang, C. and Darwiche, A.: Inference in Belief Networks: A Procedural Guide. International Journal of Approximate Reasoning 15(3) (1996) 225-263
4. Jensen, F.V., Lauritzen, S.L. and Olesen, K.G.: Bayesian Updating in Causal Probabilistic Networks by Local Computation. Computational Statistics Quarterly 4 (1990) 269-282
5. Madsen, A.L. and Jensen, F.V.: Lazy Propagation: A Junction Tree Inference Algorithm Based on Lazy Evaluation. Artificial Intelligence 113 (1-2) (1999) 203-245
6. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Francisco California (1988)
7. Silberschatz, A., Korth, H.F., and Sudarshan S.: Database System Concepts. 4th edn. McGraw-Hill Companies, New York (2002)
8. Wong, S.K.M., Butz, C.J. and Xiang, Y.: A Method for Implementing a Probabilistic Model as a Relational Database. In Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann, Montreal Quebec (1995) 556-564
9. Wong, S.K.M., Butz, C.J. and Wu, D.: On the Implication Problem for Probabilistic Conditional Independency. IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans, Vol. 30, No. 6 (2000) 785-805
10. Wong, S.K.M and Butz, C.J.: Constructing the Dependency Structure of a Multiagent Probabilistic Network. IEEE Transactions on Knowledge and Data Engineering, Vol. 13, No. 3 (2001) 395-415
11. Zhang, N.L. and Pool, D.: Exploiting Causal Independence in Bayesian network Inference. Journal of Artificial Intelligence Research 5 (1996) 301-328