A*-tree: A Structure for Storage and Modeling of Uncertain Multidimensional Arrays

Tingjian Ge
University of Kentucky
ge@cs.uky.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

Multidimensional array database systems are suited for scientific and engineering applications. Data in these applications is often uncertain and imprecise due to errors in the instruments and observations, etc. There are often correlations exhibited in the distribution of values among the cells of an array. Typically, the correlation is stronger for cells that are close to each other and weaker for cells that are far away. We devise a novel data structure, called the *A*-tree* (*multi*dimensional *A*rray *tree*), demonstrating that by taking advantage of the predictable and structured correlations of multidimensional data, we can have a more efficient way of modeling and answering queries on large-scale array data. An A*-tree is a unified model for storage and inference. The graphical model that is assumed in an A*-tree is essentially a Bayesian Network. We analyze and experimentally verify the accuracy of an A*-tree encoding of the underlying joint distribution. We also study the efficiency of query processing over A*-trees, comparing it to an alternative graphical model.

1. INTRODUCTION

Multidimensional array database systems are suited for scientific and engineering applications. Several array database systems have been designed and implemented, such as T2 [7], Titan [8], RasDaMan [2], ArrayDB [19], ASAP [24], and the most recent one, SciDB [27]. In this data model, each *cell* of a multidimensional array is a *tuple* (potentially multiple attributes, e.g., temperature, humidity, etc.). If we consider each cell's array index at each dimension as an additional attribute of the cell tuple, a multi-dimensional array is logically equivalent to a (one-dimensional) relational table. That is to say, a multidimensional array is logically equivalent to a table with schema (A_1, A_2, …, A_k , D_1, D_2, …, D_d), where each cell originally only has k attributes A_1 to A_k , and D_1 to D_d are the d dimensions of the array. Thus, an array system still follows a relational model.

Data in these applications is often uncertain and imprecise due to errors in the instruments and observations, etc. There are often correlations in the distributions of an uncertain attribute among cells of an array. For example, a temperature attribute can be correlated with other array cells (one random variable per cell). By the nature of the data that arrays model, the correlations are generally stronger for cells that are close to each other and weaker otherwise. Consider the following example.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 1
© 2010 VLDB Endowment 2150-8097/10/09... $10.00

Example 1. *Environment monitoring in an open field often produces sensor readings over time. However, resource constraints (e.g., sensor power and network capability) often prevent sensors from sending readings at every point in time. At a given time, we may only have an outdated reading for each sensor in the network. The actual current reading is from a distribution in a range around the outdated one* [9]*. All sensors (which can be either real sensors or interpolated readings* [15]*) in the network form a two-dimensional array with the dimensions being sensor location. The uncertain attribute is correlated with neighboring cells.*

We provide two additional examples in Appendix A. In many such applications, the representation of the uncertain data needs to encode the value correlation among tuples for the result to be correct. Ignoring the correlation and making an over-simplified tuple independence assumption often renders the query results wrong and useless. We illustrate this in the experiment section.

However, modeling attribute correlation among cell tuples is not an easy task, simply due to the large number of cells in many arrays and the arbitrary correlations among them. In this work, we argue that by taking advantage of predictable and structured correlations of multidimensional data, we can provide a more efficient way of modeling and answering queries on large-scale array data. We propose a new data structure, called the *A*-tree* (*multi*dimensional *A*rray *tree*). The A*-tree approach is based on the following observation: data in a multidimensional array is usually correlated along some dimensions and the correlation is largely local. Thus, if we have to sacrifice precision by allowing approximate models, we should focus on local correlation. An A*-tree uses this fact and organize data in a hierarchical manner. Within the hierarchical structure, the joint distributions are smaller and can be modeled more efficiently.

There is a simple mapping from the *graph structure* of an A*-tree (i.e., the storage model of an array) to its *probabilistic graphical model*. We show that the graphical model of an A*-tree is a Bayesian Network (BN). Physically, only the leaves of the tree-structured BN exist. The nodes (i.e., random variables) at upper levels are *derived* from the leaves. Thus, the construction of an A*-tree is bottom-up, yet the probabilistic inference (which is needed for processing queries [22, 25]) is top-down.

Because A*-tree integrates both the probabilistic graphical model and the physical storage model, probabilistic inference is very efficient. It requires traversing the A*-tree and following a logarithmic-length path directly to the needed cells of the array. We study query processing techniques for both general queries and specifically COUNT, AVG, and SUM queries which permit optimizations using A*-trees. We also compare A*-trees with an alternative graphical model. To summarize, the paper's contributions are:

- We propose a novel data structure, A*-tree, suited to modeling the correlation in a multidimensional array (Sec. 2).

- We analyze how A*-trees balance the accuracy of modeling correlation and the efficiency of query processing (Sec. 3).
- We study query processing techniques, demonstrating the advantage of efficient inference with A*-trees (Sec. 4).
- We conduct a systematic experimental study on a real dataset as well as a synthetic dataset (Sec. 5).

2. A*-TREE STRUCTURE

In this section, we describe the A*-tree structure and how it encodes the joint distribution of array cells.

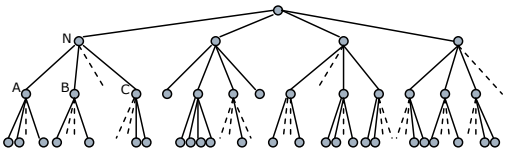


Figure 1. Example of a *k*-ary tree with *k* = 4

2.1 Basic A*-tree Structure

An A*-tree is a *k-ary tree* [11] (e.g., Figure 1, where dotted branches indicate which children are missing) with the degree k being 2^d, where d is the number of dimensions in which the uncertain value is correlated. Note that d is typically small (most often 1, 2, or 3). Thus, it is a binary tree when $d = 1$ and a 4-ary tree when $d = 2$, and so on. Figure 2 shows an example partition for $d = 2$. Throughout this section, we use $d = 2$. This can be easily extended to other dimensionalities. Similar to quadtrees [16], we recursively divide an array in half along each dimension. In Figure 2, the first partition (thick dotted lines) divides the array space into four ($k = 2^2$) subspaces. The whole array maps to the root of the 4-ary tree in Figure 1, and the four subspaces map to its four children in some fixed order (e.g., 1st child is the north-west subspace, 2nd child is the south-west one, etc.). Then recursively, we again partition each of the four subspaces into four, which map to the four children of each node at the level below the root in Figure 1. Thus, a recursive partition of the array space corresponds to a top-down traversal of the k-ary A*-tree from one level to the next. Eventually, at the leaf level, each leaf corresponds to four neighboring cell values of the array. In Figure 2, array cells A, B, C, and D together form a leaf.

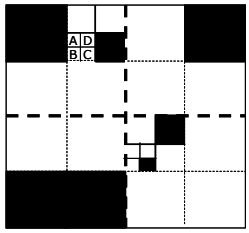


Figure 2. Illustrating recursive partitioning of a two-dimensional array. The joint distribution of the uncertain attribute is encoded in a 4-ary tree.

For now, for simplicity of exposition, in the case of $d > 1$, we assume that each of the dimensions has the same size. We also assume this size is 2^n (for some integer n). It is easy to generalize it to an arbitrary size (as shown in Appendix D.1). The black blocks in Figure 2 indicate the *empty* regions (NULL values) of the array that do not have values in the A*-tree and, thus, correspond to "missing" children in this 4-ary A*-tree. Thus, arrays of arbitrary sparsity can be accommodated. Here is how a joint distribution is encoded in an A*-tree:

- Each leaf in an A*-tree contains the joint distribution of four neighboring cells of the array. This joint distribution is a conditional one, conditioned on the four cell's average value. In the example in Fig. 2, there are four cell random variables A, B, C, and D. Define a random variable $X = (A+B+C+D)/4$. Then there is a leaf in the A*-tree that contains the joint distribution of A, B, C and D conditioned on X.
- Recursively, in a bottom-up manner, an internal node of the A*-tree encodes the joint distribution of its four children conditioned on their average. A child node is a random variable that is the average of all cells of the array covered under its subtree.
- In addition to this joint distribution, the root node also holds the distribution of the average value of the whole array.

Note that the average of children is *weighted*. For example, in the A*-tree of Figure 1, node N contains the joint distribution of its three children (one child node is missing), A, B, and C, conditioned on their average value $(n_A \cdot A + n_B \cdot B + n_C \cdot C) / (n_A + n_B + n_C)$, where n_A is the number of non-empty cells (i.e., not NULL) in the subtree rooted at A; similarly for n_B and n_C.

The key idea of A*-trees is that we model the joint distribution of cells in a manageable way that is relatively compact and automatically structured. The automatic structure is based on the principle of the locality of data correlation: closer cells are more likely correlated. We organize cells into hierarchical clusters according to proximity, each of which contains a small number of random variables so that we can encode their joint distribution compactly.

An interesting aspect of the A*-tree approach is that if we simply trim the leaves of an A*-tree, the remaining A*-tree represents the distribution of an array with a coarser grain. This enables a fast approximation of the data and may be meaningful for many applications that demand rapid results (e.g., real-time processing or on-line computation). For the example of image and sound, object or pattern recognition algorithms can work in the coarser level. In a real-time network system, quick decisions at a higher level can be crucial for meeting real-time constraints.

Finally, we note that A*-trees partition the space in an array similar to the way quadtrees [16] do (for two dimensions). Our main contribution in this work, however, is about using this partition scheme to succinctly model the correlations and joint distribution of the array cells. We show that this is a natural unification of both the storage model and the probabilistic graphical model (Section 3).

2.2 Extensions of the Basic A*-tree Structure

2.2.1 Basic Uncertainty Blocks of Arbitrary Shapes

We define a basic uncertainty block of an array as a box (e.g., a rectangle for two-dimensional arrays) in the array inside which cells have the same distribution. In the basic A*-tree of Section 2.1, each array cell is a basic uncertainty block. This is the smallest basic block size possible. However, in many applications, this granularity is not necessary and the basic block size can be much larger. Having a larger basic block size makes the representation more succinct and query processing more efficient.

For example, astronomers take photo images of objects in the universe. Due to precision limits, pixels of an image, treated as cells of a two-dimensional array, exhibit correlated uncertainty in their values. A block of neighboring pixels, due to their proximity, is likely to have the same error distribution. Thus, a basic uncertainty block can be, say, 50 by 50 cells in size. Now each basic block is treated as a "single cell" in an A*-tree, which

only records a single distribution. Each basic block will also store a 50 by 50 block containing the "deterministic" parts of the pixel values. Combining the deterministic and the random parts together gives a true pixel value.

2.2.2 Initial Partition of an Array

The best initial partition of an array is application specific and a knowledgeable user can define the initial partitions. In the astronomy's image example, different regions of the image may have different levels of uncertainty. Some parts of the image (e.g., towards the center) are clearer and have less uncertainty, while some parts (e.g., towards the borders) are blurrier and have more uncertainty. Thus, one may want to first partition the array into rectangular regions and assign different basic block sizes for different regions: regions towards the image center have larger basic uncertainty blocks and the distributions there have smaller variances, while regions at the borders need finer basic blocks. This is illustrated in Figure 3.

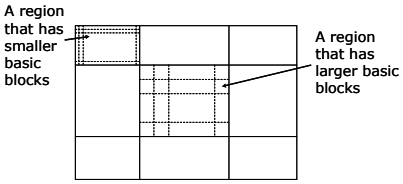


Fig. 3 Illustrating the initial partition of an array into nine regions.

Since correlation among regions may be very weak, an application program can either declare region summaries (i.e., average values) to be independent or let the system manage the joint distribution of the regions as the upper levels of the A*-tree. When regions are independent, each of them is a separate A*-tree.

We provide additional details of A*-trees in Appendix D, such as the joint distributions at nodes and their layout on disk.

3. ANALYSIS

3.1 A*-tree's Probabilistic Graphical Model

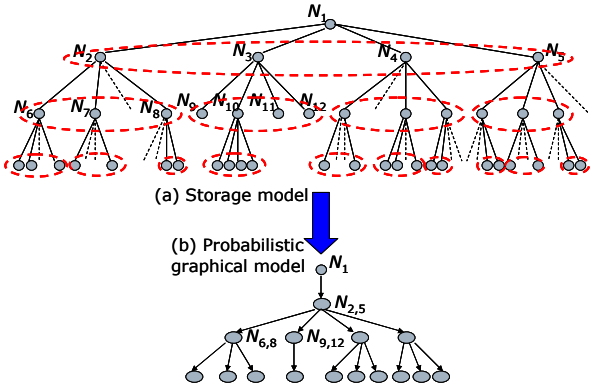


Fig. 4 An A*-tree encodes a unified storage model (a) and probabilistic graphical model (b). There is a natural conversion from (a) to (b). The root node (N_1) stays unchanged. Shrink its children (N_2 to N_5) into one node, indicated by the dotted ellipse in (a) and $N_{2,5}$ in (b). The four edges connecting N_1 with N_2 … N_5 in (a) are shrunk into one directed edge in (b). The similar procedure applies to other nodes and we get a Bayesian Network in (b).

An A*-tree is a unified structure for both the storage model and the probabilistic graphical model. Some background knowledge on graphical models is presented in Appendix B. An A*-tree is a special form of Baysian Network (BN) on the array cell values, as illustrated in Figure 4. There is a natural mapping from the storage model of an A*-tree (Figure 4a) to its graphical model (Figure 4b). In a nutshell, we need to collapse the multiple children of an internal node (e.g., N_2 to N_5 in Figure 4a) into one *composite* node in the graphical model (denoted as $N_{2,5}$); corresponding edges are also merged. This is needed because an A*-tree encode $P(N_{2,5}|N_1)$, but not $P(N_2|N_1)$, etc. Recall that given a value at N_1, an A*-tree gives us the joint distribution of nodes N_2, N_3, N_4, and N_5. Likewise, each edge in Figure 4b corresponds to a joint distribution in a node of the A*-tree. Note that a node in the graphical model of an A*-tree can be *composite*, denoting several nodes of the A*-tree.

The unconventional aspect of this BN is that originally *only the leaf level exists* and represents *real* random variables (each leaf maps to some cells of the array). All internal nodes (random variables) are *artifacts* of our construction. They are *derived* random variables. Because the internal nodes are completely determined by the values of the leaves (i.e., they are averages at different levels), the distribution encoded by the whole BN is equivalent to the joint distribution of the array cell values.

3.2 Expressiveness of Neighbor Correlation

An A*-tree expresses neighboring correlations in the joint distributions at different levels of the tree. Clearly, the correlation between two cells is easier to encode when this level is lower. We demonstrate that, from the perspective of any random query, the average level where cell correlation is encoded is low.

Definition 1 (*neighboring cells* and *cluster distance*): Two *neighboring cells* of an array are two cells that are next to each other in one dimension and have the same dimension values in other dimensions. Starting from the cell level (leaves) as level 0, the *cluster distance (CD)* between two neighboring cells is the level in the A*-tree at which a joint distribution between their cluster summaries exists. □

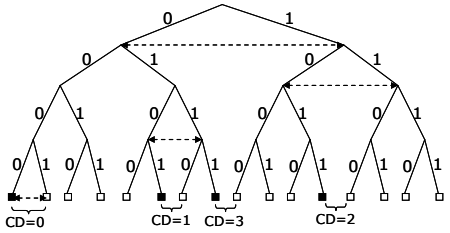


Fig. 5 Illustrating cluster distance in a binary A*-tree

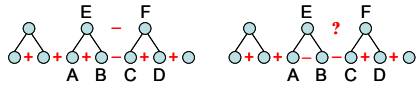
Figure 5 shows an example of pairs of neighboring cells with CD 0, 1, 2, and 3, respectively. We can see that the CD between two cells is determined by the level below their *lowest common ancestors*. When CD is 0, the correlation between two cells is directly modeled; when CD gets bigger, their correlation is embodied in the summaries of bigger clusters they are in. We next quantify the *average* as well as the *maximum* CD in the set of cells that an arbitrary query accesses.

Lemma 1: *For d=1, we label every left branch of a binary A*-tree with* 0 *and every right branch with* 1. *We then label each cell of the array with the concatenation of labels on the path from root to the cell. Then the CD between a cell and its right neighbor is simply the number of trailing* 1*'s in its label.*

Theorem 1: *Consider a binary A*-tree (d = 1) of height h. Suppose a query references a random part of the array that has q pairs of neighboring cells (either in a contiguous range or scattered in the array). Then the expected average CD is*

$1+(h+1)/2^h$ *and the expected maximum CD of the q pairs is* $\log q + 1 - q/2^{h-1}$. *For d =2 (4-ary A*-tree), the expected average CD is the same and the expected maximum CD is* $\log q + (q_1+q_2)\left[\frac{1}{q} - \left(\frac{1}{2}\right)^{h-1}\right] - \frac{q_1 q_2}{3}\left[\frac{1}{q^2} - \left(\frac{1}{4}\right)^{h-1}\right]$ *where* q_1 *and* q_2 *are the number of neighboring pairs along the two dimensions and q = max(*q_1*,* q_2*).*

The proofs of all theorems in the paper are in Appendix C. In the same vein, we can obtain the CD's for larger *d* values. Theorem 1 implies that, from the perspective of any random incoming query, the expected average cluster distance of neighboring cells is very close to 1 for any reasonable size array in practice. For instance, for a 1024 × 1024 array, h = 10, the expected average CD is about 1.01. We next show the underlying intuition for why this result indicates that neighboring correlations are well modeled in practical array data.



(a) (b)

Fig. 6 (a) "single flip" and (b) "double flip" in A*-trees

The correlation between two neighboring cells, if any, is either positive or negative. Consider a sequence of pair-wise correlations along one dimension, as shown in Figure 6a. This is a piece of an A*-tree. The bottom level corresponds to array cell values. "+" denotes a positive correlation between two neighboring random variables while "−" denotes a negative correlation. Two or more *consecutive* negative correlations in an array (between A and B then between B and C in Figure 6b) is called a *double flip* (a negative correlation is analogous to a "flip": when one value increases, the other one decreases). In general, positive correlations are much more common and negative correlations which we call a *single flip* (between *B* and *C* in Figure 6a), if present, are usually isolated.

In Figure 6b, the second level nodes E and F may not exhibit negative correlation: when F and C increases, B tends to decrease and A tends to increase which implies that E (the average of A and B) could change in either direction or stay unchanged. Thus, in this case, the negative correlation between B and C cannot be modeled accurately in the second level of the A*-tree. However, we observe that for most array data in practice (e.g., image data or environmental properties such as temperatures), a double-flip is very rare. *Even if it were not rare*, we could easily locate any double flips by a single pass of the array (with a cost $O(dn)$, where *d* is the dimensionality and *n* is the number of cells of the array). For a double flip, the solution is to model it separately with a joint distribution on A, B, and C as a *basic uncertainty block* as discussed in Section 2.2. One can verify that this is not an issue with Fig. 6a, where a negative correlation can either be expressed in the first or the second level.

This observation, combined with the result of Theorem 1 that the expected average CD is very close to 1 (i.e., correlations are expressed in low levels), explains why the loss of the modeling accuracy for correlation is generally not a serious issue in practice, which we also experimentally verify in Section 5. Our gain from this tradeoff is the efficiency and simplicity in modeling and query processing, as we demonstrate throughout the paper.

4. QUERY PROCESSING

In this section we discuss techniques of doing query processing on multidimensional arrays with uncertain attributes represented as A*-trees. We first look at processing general queries and then consider optimizations for COUNT, AVG, and SUM queries.

4.1 Queries in General

Scientific applications are often computationally intensive and tend to use a different set of operators (e.g., dot products, matrix multiplications). The design of an array database system must take these operators into consideration [24, 19, 2, 7]. The complex nature of the query operators complicates the task of probabilistic inference with graphical models. Consequently, often the most viable method of probabilistic inference is through Monte Carlo (MC) algorithms [5]. Query processing through MC involves multiple rounds. In each round we obtain one sample for each probability distribution. Then any classical database system can run the query over these samples. Finally, we "assemble" the query results from these multiple rounds together and form the result distributions. A significant advantage of using MC for query processing is that it can essentially process *any* type of query on uncertain data [17, 18] under possible world semantics. All four types of probabilistic queries as classified in [9] can be answered. Specifically, a distribution in the result of a *value-based query* is learned from the result values of multi-rounds of MC, while a tuple probability in the result of an *entity-based query* is based on the frequency of the tuple appearing in the results of multi-rounds. Thus, we first describe the sampling algorithm from an A*-tree given an incoming query (so that we can use MC for any type of query). We then demonstrate its efficiency by comparing with the alternative MRF (Markov Random Fields) models.

4.1.1 Sampling

Sampling from an A*-tree is an efficient top-down traversal (logarithmic-length path), shown in Figure 7. It is an application of the ancestral sampling technique [5] on the Bayesian Network in Figure 4(b). The tree structure allows us to limit the sampling to the path from the root to the target cells Q, and nothing else. Note that from the recursive partition of the array dimensions during A*-tree construction, it is easy to determine the range of dimension values associated with each node. Step (6) in the algorithm uses such information to determine if there is an overlap between the coverage of a node and the set Q.

Input: An A*-tree *T*, a set of cells *Q* accessed by a query.
Output: A set of samples *S*, one value for each cell in *Q*, from the joint distribution of *T*.

(1) At the root of *T*, from the *distribution of the average value of the whole array*, get a sample for the root.
(2) Initialize node set N = {root} (one node).
(3) **For** each node $n \in N$,
(4) Sample from the joint distribution at *n*, get sample values (v_1, v_2, v_3, v_4) for its four children, based on the sample at node *n*.
(5) **If** *n* is a leaf of *T*, then v_i ($1 \le i \le 4$) is for a cell *c*. If $c \in Q$, then v_i is the final sample for *c*.
(6) **Else** for each child c_i ($1 \le i \le 4$), if the range of dimension values covered by c_i intersects *Q*, then add c_i to *N*.
(7) **End for**

Fig. 7 An algorithm to get samples for a set of array cells.

As an example, consider the following astronomy query:

***Q*1: SELECT AVG**(*brightness*) **FROM** *Space_image*
WHERE DISTANCE(*x*, *y*, *z*, 322, 108, 251) < 50

*Q*1 asks for the average *brightness* within a certain distance (50) of an object at location (322, 108, 251). *Space_image* is an array with three dimensions *x*, *y*, and *z* indicating locations of objects. An A*-tree is built on the *brightness* attribute. DISTANCE is a built-in function that calculates the distance between two locations. The most effective known method of probabilistic inference for such a query on a graphical model is based on MC algorithms [5]. Our system optimizer will compute a minimum bounding box that contains the sphere selected by the WHERE clause. The bounding box is a first approximation of the set of cells *Q*, as input to the sampling algorithm in Figure 7. The algorithm starts from the root and traverses down the tree, targeting only the bounding box *Q*, which is eventually refined to the actual sphere required by the WHERE clause. Note that our optimizer will obtain all the samples of a cell needed by MC (say, 100 samples) at the same time because they are independent. Thus, we only need to traverse down the tree once, thereby saving I/O costs. This is in contrast to sampling from MRF (Section 4.1.2), in which we cannot use this optimization because sample rounds are correlated and must occur in sequential order.

4.1.2 Comparison with an MRF model

We give some background on MRF in Appendix B. One may wonder what would result if we just model a multi-dimensional array with a simple lattice structure MRF to capture the neighborhood correlation, as shown in Figure 8(a). However, the problem here is the high computational cost. How big is the MRF model? Ideally, it should span the whole array so that all the local correlations between all pairs of neighboring cells are captured by the model. However, the computation cost of sampling a big MRF is high, as we illustrate next.

The corresponding inference algorithm for an MRF is Markov Chain Monte Carlo (MCMC) [5]. Gibbs sampling [5] is often used with MCMC on an MRF. Each node is sampled from its distribution conditioned on its neighbors. Each sampling round updates the samples of all nodes. It has to iterate though *all* the nodes in a model to create one sample, even though the query may only need to access a tiny fraction of the cells of the whole array. Gibbs sampling uses a so-called *visitation schedule* to update the samples of each node in the graph to obtain one sample from their joint distribution. This is because all nodes are either directly or indirectly connected, and thus the sample value of each node is needed to produce the next round of samples. Therefore, the sampling is rather wasteful for answering a query.

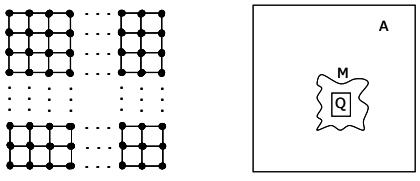


Fig. 8 Illustrating MRF construction for a two dimensional array. (a) indicates a simple grid structure. (b) illustrates a box Q actually needed for answering a query inside array A. An MRF over an arbitrary region M that contains Q is used.

Now suppose we do not use an MRF model for the whole array A. Instead, we have an MRF model built over a small region M (of any shape) inside A and M contains Q, the set of cells accessed by the query. This is illustrated in Figure 8(b). We could use the model over M to give an approximate answer to the query. However, the area Q accessed by some incoming query can be arbitrary, and it would be impractical to dynamically build (learn) a model on the fly at execution time or to have a sufficient number of pre-built models.

By contrast, our A*-tree sampling algorithm is based on *ancestral sampling* [5] over Bayesian Networks (which is basically sampling in the order specified by the directed edges, i.e., always sample parents before any children). As described in the sampling algorithm in Figure 7, our sampling procedure only follows a logrithmic path from the root of an A*-tree to the set of leaves used by a query.

Furthermore, the ancestral sampling of A*-trees is much more efficient than MCMC sampling for MRF. MCMC requires a *mixing time* before its samples can be used (i.e., the Markov chain needs to get to a stationary distribution first; a.k.a. "*burn in period*") [5]. Rigorous justification of inference results would require a theoretical bound on mixing time, and many interesting practical cases have resisted such theoretical analysis [5]. A Markov chain may converge very slowly to its stationary distribution, requiring a long mixing time. In Section 5, we further experimentally study the impact of the mixing time of MRFs on result accuracy and speed.

Finally, MCMC sampling requires the samples to be correlated (forming a Markov chain) and in a serial order. As a result, we cannot use the optimization of performing all sampling rounds concurrently to save I/O costs as we did for A*-trees (see end of Section 4.1.1). For example, in answering *Q*1 (Sec 4.1.1), the system needs to follow the site visitation schedule and perform sample rounds one by one (each round obtains one sample for each cell in the bounding box *Q*).

4.2 COUNT, AVG, and SUM Queries

For sparse arrays, applications often query the COUNT, AVG, or SUM of "non-empty" cells (i.e., with a value in the A*-tree) that fall within in a *bounding box* (i.e., a range in each dimension). It turns out that we can answer these queries very efficiently using the A*-tree data structure.

We add an integer value (*cell_count*) to each internal node of an A*-tree, recording how many non-empty cells there are in the subtree rooted at the node. The *cell_count* of all nodes can be easily obtained in a bottom-up manner during the construction of the A*-tree. Next we introduce a definition.

Definition 2 (*minimum cover*): A *minimum cover* of a set of cells of an array is a set of nodes in an A*-tree whose subtrees contain exactly the set of cells (no more and no less). Further, there does not exist another set of nodes that has this property but with fewer nodes in it. □

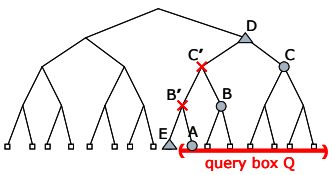


Fig. 9 Illustrating minimum cover and minimum cover with subtraction

For example, in Figure 9, the minimum cover of the query bounding box *Q* (last seven leaves, or cells in the array) has three nodes: A, B, and C. Clearly, once we have the minimum cover of cells in a bounding box, adding up the *cell_count* in all nodes in

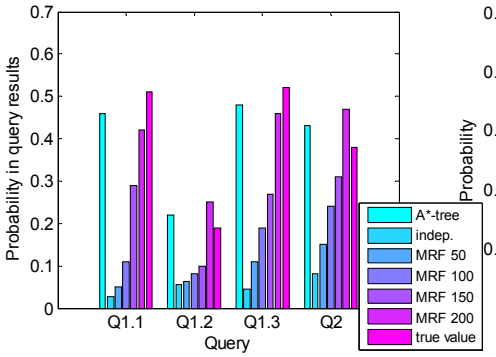


Fig. 10 Q1 & Q2's results.

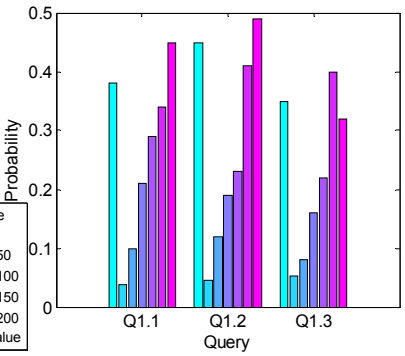


Fig. 11 Q1's results with the synthetic dataset.

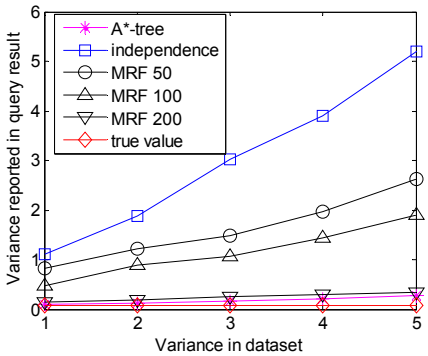


Fig. 12 Q3's results for various methods

the minimum cover gives us the COUNT of non-empty cells. This means that during query processing we can *stop early* at the minimum cover without going further down the tree. Nonetheless, one might wonder if this is the best we can do. In Figure 9, for example, we could also use nodes D and E (*cell_count* in D minus that in E), which uses one fewer node. As we increase the tree height, the difference gets bigger. We call such a node set a *minimum cover with subtraction*. However, the following theorem shows that it does not really reduce the access cost.

Theorem 2: *In an A*-tree stored on disk in level-order (*[4]*, also Appendix D.3), for a group of sibling nodes, assume that accessing any non-empty subset of them incurs the same I/O cost (since they are stored contiguously). Then accessing a minimum cover has the same I/O cost as accessing the corresponding minimum cover with subtraction.*

For an A*-tree, we can easily find out, for each node, the range in each dimension of the array that it covers. Thus, the algorithm to compute the minimum cover *MC* for a set of cells *Q* is quite simple: Starting from the root, we check if the node covers only cells in *Q*. If so, we add this node to *MC*; otherwise we recursively check each of its children that has an overlap with *Q*.

For a COUNT of non-empty cells, we simply add up the *cell_count* in the nodes of *MC* and do not need to do anything extra. For AVG and SUM queries, however, we need to combine with the sampling technique described in Section 4.1.1. In Monte Carlo query processing, the sampling would be done together with our top-down procedure above to get an *MC*. Then we *stop early* at nodes in *MC* without sampling further down the tree. Let the sample value and *cell_count* at each node in *MC* be a_i and c_i, respectively ($1 \le i \le t$, where t is the cardinality of *MC*). Then the SUM and AVG for this sampling round are $\sum_{i=1}^{t} c_i a_i$ and $(\sum_{i=1}^{t} c_i a_i) \Big/ \sum_{i=1}^{t} c_i$. Thus, for queries over large-scale datasets, many nodes in MC are at high levels and our optimization can significantly improve the performance.

5. EXPERIMENTS

5.1 Accuracy of Modeling the Underlying Joint Distribution

We describe the datasets and the setup of experiments in Appendix E. The Intel Lab dataset contains sensor readings that span about 65,535 epochs. We use the temperature readings from that dataset. An epoch is a monotonically increasing sequence number from each sensor. Two readings from the same epoch number were produced from different sensors at the same time. Temperature readings at missing time instances can be inferred and are uncertain. We use A*-trees to model the inferred readings at missing time points. This uncertain data forms a three-dimensional array with the first two dimentions being the location in the lab and the third dimension being time. At each missing time instance, we have a grid of temperature values, some of which are missing. Using linear interpolation [15] from neighboring cells we can add more temperature values.

The joint distribution at each node of an A*-tree is learned from a short period of time (100 epochs). In order to test if the A*-trees model correlations correctly, we first query the existing dataset and find three groups of sensors that have a relatively high frequency, during all 65,535 epochs, of temperature readings within a range of one degree. Each group has four sensors. The first group has sensors at locations (2, 27), (11, 24), (6, 32), (6, 33) in the grid and the second group has sensors at (60, 2), (60, 3), (61, 2), (61, 3), etc. The *x* and *y* coordinates of sensors are in meters relative to the upper right corner of the lab space. We then arbitrarily pick an A*-tree and query the probability that a group of sensors has close temperature readings (within one degree):

```
Q1: SELECT close_values (temperature, 1)
    FROM lab_array
    WHERE (x = 2 AND y = 27) OR (x = 11 AND y = 24)
          OR (x = 6 AND y = 32) OR (x = 6 AND y = 33)
```

Q1 is on the first group of sensors. *close_values* is a user-defined aggregate that takes a set of temperature attribute values as the first parameter, and returns 1 if the set of values are all within a distance range of each other (1 degree in the above query). Thus, using Monte Carlo query processing, we can compute the probability that the result is 1, which is the estimated probability that the group of four sensors have close values.

We show Q1's results for the three groups of sensors at epoch 800 in Figure 10 (Q1.1, Q1.2 and Q1.3). We retrieve 50 samples from the A*-tree and compute the resulting probability. We execute the query for each sensor group. To compare with the result from an alternative graphical model of a lattice structured MRF, we build an MRF for each of the four sensor groups, as illustrated in Figure 8. Using Gibbs sampling and MCMC [5], we compute the results of the four queries. As discussed in Section 4.1.2, MRF is *impractical* if we do not know the queries in advance. Thus, here we assume that the system does know what queries will be asked ahead of time and builds suitable MRF models over small regions of the array ahead of time. Note that a salient advantage of A*-trees is that we do not need such an assumption; the system just builds one structure and accesses different parts of it according to various queries.

For comparison, we also use the first 50 samples of MRF, as in A*-trees. As discussed in Sec. 4.1.2, due to the *mixing time* of

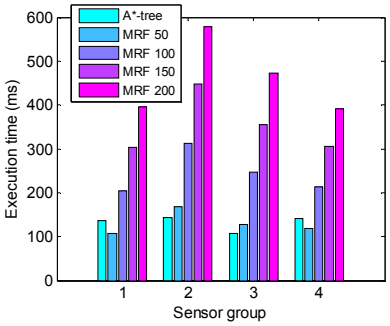


Fig. 13 Execution time comparison.

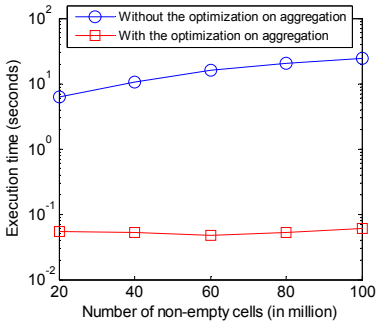


Fig. 14 Execution time of an aggregate query.

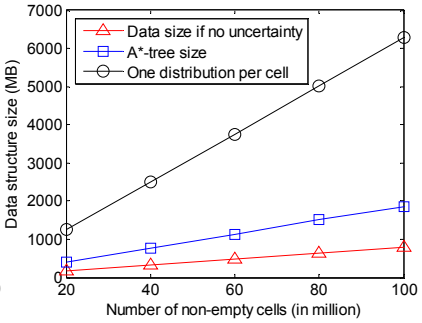


Fig. 15 Examining A*-tree size.

MRF sampling, the initial samples are not from the stationary distribution and thus are not of good quality. Therefore, we also experiment with 100, 150, and 200 samples respectively, but only use the final 50 samples to compute the result. We omit the initial samples in order to pass the mixing time, and always use the last 50 samples for comparisons. We also compute the result under the independence assumption (the second bar). Finally, we also compare these results with the statistics collected over all epochs in the dataset (the last bar), which serve as an indication of the underlying *true* joint distribution (i.e., the true query result).

From Figure 10, we can clearly see that A*-trees model the underlying joint distribution very well in terms of the accuracy of inference results. On the other hand, the approach based on the independence assumption produces a very small probability result because it does not model the correlation among the sensors and thus, the probability that all four independent sensor samples are close to each other is small. The fact that we arrive at the correct results with A*-trees verifies the well-structured correlation of the data. For the Markov chain sampling from MRF's, we can see that because of the mixing time, it slowly converge to a stationary distribution. For the same number of sampling rounds, A*-trees give much better results. MRF can catch up with more rounds, but that takes much longer time (as we show in Section 5.2). We next repeat this experiment with the synthetic dataset. Again we use three groups of sensors at different locations. The result is shown in Figure 11 (which shares the same legend as Figure 10). This dataset again verifies our observations earlier.

We use two other queries (Q2 & Q3) to further verify our arguments. Q2 and Q3 access a larger and different part of the array than Q1. In the Intel Lab, there is a "server" room. Imagine that the lab administrator would like to be sure that various locations in the server room maintain a constant temperature for the benefit of the machines. In the dataset, the server room's location is approximately at the rectangle from the upper-right position (12.5, 18) to the lower-left position (19, 25.5). She could arbitrarily pick three random points in the server room and ask this query:

***Q2: SELECT** 1 **FROM** lab_array*
* **WHERE** x=14.5 AND y=24 AND*
* temperature BETWEEN (18, 19)*
* **INTERSECT***
* **SELECT** 1 **FROM** lab_array*
* **WHERE** x=17.5 AND y=19.5 AND*
* temperature BETWEEN (18, 19)*
* **INTERSECT***
* **SELECT** 1 **FROM** lab_array*
* **WHERE** x=16 AND y=22 AND*
* temperature BETWEEN (18, 19)*

Q2 essentially asks for the probability that three random locations in the room (14.5, 24), (17.5, 19.5), and (16, 22) all have temperatures between 18 and 19 degrees Celsius. The dummy constant tuple "1" is in the result with some probability and our system that manages uncertain data returns this probability (result tuple uncertainty). The result of Q2 is shown in the final group of bars in Figure 10. The result and reasons are similar to Q1's. We then issue the following query:

***Q3: SELECT** variance (temperature)*
* **FROM** lab_array*
* **WHERE** x BETWEEN (12.5, 19) AND*
* y BETWEEN (18, 25.5)*

The user-defined aggregate "*variance*" takes a set of *temperature* attribute values as parameters, and returns their variance. When the temperature is relatively constant across various locations of the room, this variance should be small. We vary the variance parameter in the uncertain data which we specified earlier and execute the query. The results are shown in Figure 12. When each temperature point is modeled independently, because the expected values of the temperature variables across the room are very close, the variance returned by the query is close to each individual point's variance. However, the temperature variables are actually correlated. Thus, the correct result of Q3 should have a very small value as indicated by the statistics collected over all epochs (which is a constant). The results of Q2 and Q3 once again confirm our findings that show that A*-trees produce much better results than using the independence assumption or using MRF with the same number of rounds. MRF with 200 rounds has almost comparable result accuracy with A*-trees, but, as we show in the next subsection, it is much slower, in addition to the fact that MRF requires the assumption that query workloads are known in advance.

5.2 Execution Time

We now examine the execution time for answering the queries in Section 5.1. The result of answering Q1 on the lab dataset is shown in Figure 13 (Q2 and Q3 show similar comparisons). We measure the execution time of answering the query by generating 50 samples from the A*-tree. We also measure the execution time by generating 50, 100, 150, and 200 samples from MRF's. We can see that using MRF's is significantly slower than using A*-trees in order to provide a result that has about the same accuracy.

5.3 Aggregate Queries

We now examine the performance improvement of the optimization using the *minimum cover* for COUNT, AVG, and SUM queries presented in Section 4.2. To arbitrarily control the data size, we use the synthetic dataset whose schema is the same as the Intel Lab dataset. We can programmatically control both the size of the array and the fraction of empty cells in the array. The array size is 32K by 64K (i.e., 2G cells) with half of them empty. We issue an aggregate query of the following form:

Q4: SELECT *AVG(temperature)*
FROM *synthetic_array*
WHERE *x BETWEEN ? AND ?*
AND y BETWEEN ? AND ?

By controlling the parameters, we run Q4 over different numbers of non-empty cells. We compare the running times with and without the optimization presented in Section 4.2. In both cases, we perform 300 concurrent rounds of sampling whenever we get to a node of the A*-tree. This avoids going back to the node again and saves I/O costs. Figure 14 shows the comparison. We use a log scale on the *y*-axis of Figure 14 in order to show both lines clearly. The optimization is about two orders of magnitude faster because it only accesses the A*-tree nodes on the path from the root down to the *minimum cover*, instead of accessing nodes all the way down to the leaves (as is the case without the optimization).

5.4 Space Consumption

Using the generated synthetic dataset, we examine the space costs of A*-trees. Figure 15 shows the details. The *x*-axis indicates the number of non-empty cells of two-dimensional arrays with different sizes in which about half of the cells are empty. We compare the sizes of the A*-trees with an obvious lower bound in which the data has no uncertainty at all. Figure 15 shows that the A*-tree sizes are a little more than twice the lower bound. We also compare with a naive approach in which an array stores one distribution per non-empty cell. This does not model the correlation between cells, and the sizes of the resulting arrays are significantly bigger than A*-trees. Note that a lattice-structure MRF model for the *whole* array, which is too costly for query processing, would have a similar size because we need to store, at each cell, the conditional distribution of the cell on its neighbors for sampling. We also note that the space consumption for A*-trees can be further reduced when the *basic uncertainty blocks* are bigger than single cells, as discussed in Section 2.2.

6. RELATED WORK

Uncertain data management has not been studied in previous array systems, although uncertainty is common in scientific data. There has been extensive work on managing uncertainty in traditional databases, e.g., [12, 3, 9, 1, 23]. Probabilistic graphical models have been used in databases to model correlation. For example, Sen and Deshpande [22] are among the earliest. Wang et al. [25] propose a declarative relational extension of BN models to capture correlations at various levels of granularity. Our work, however, takes advantage of the predictable and structured correlations present in multidimensional data. We can provide a more efficient way of representing uncertainty in large-scale array data and of answering queries over this data.

Note that several techniques in our work have similarity to other work in different contexts. The space partitioning scheme in A*-tree has been used before. For example, Dasu et al. [13] use it for the task of change detection. Dealing with averages in a tree is similar in spirit to multi-resolution synopsis structures like Haar wavelets [6]. The machine learning community has studied ways to simplify a general model for efficient inference (e.g., [10, 14]). However, our techniques are fundamentally different in that (1) we use "auxiliary variables" – all internal nodes of an A*-tree represent functions (i.e., averages) of the *leaves*; and (2) we take advantage of the array structure. Finally, there has been recent interest in applying Monte Carlo algorithms for managing uncertain data (e.g., [21, 17, 18]).

7. CONCLUSIONS AND FUTURE WORK

Correlations are common in array data and they are structured along dimensions. Based on this observation, we develop a novel data structure, called A*-tree, which is a unified model for storage and modeling of such data. We demonstrate that compared to alternative approaches, an A*-tree can not only perform inference much more efficiently, but it also models the underlying joint distribution accurately. A systematic empirical study is conducted on both real and synthetic datasets. As future work, we plan to evaluate A*-trees in the domains of astronomy or biology, where datasets tend to be larger, or where scenarios such as time-varying data or hardly-correlated data may arise.

8. REFERENCES

[1] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and Simple Relational Processing of Uncertain Data. In *ICDE*, 2008.
[2] Baumann P, Dehmel A, Furtado P, Ritsch R, Widmann N. The multidimensional database system RasDaMan. In *SIGMOD*, 1998.
[3] Benjelloun, O., Das Sarma, A., Halevy, A. and Widom, J. ULDBs: Databases with Uncertainty and Lineage. In *VLDB*, 2006.
[4] D. Benoit, E. Demaine, J. Munro, R. Raman, V. Raman, and S. Rao. Representing Trees of Higher Degree. In *Algorithmica*, v.43, 2005.
[5] C. Bishop. *Pattern Recognition and Machine Learning*, 2006.
[6] K. Chakrabarti, M. Garofalakis, R. Rastogi, K. Shim. Approximate query processing using wavelets. In *VLDB Journal*, 2001.
[7] C. Chang, A. Acharya, A. Sussman, J. Saltz. T2: a customizable parallel database for multi-dimensional data. In *SIGMOD*, 1998.
[8] Chang C, Moon B, Acharya A, Shock C, Sussman A, Saltz JH. Titan: a high-performance remote sensing database. In *ICDE*, 1997.
[9] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.
[10] A. Choi, H. Chan, A. Darwiche. On Bayesian Network Approximation by Edge Deletion. In *UAI*, 2005.
[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms* (2nd edition). MIT Press and McGraw-Hill.
[12] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.
[13] T. Dasu et al. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *Interface* 2006.
[14] R. Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference. In *UAI*, 1996.
[15] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.
[16] R. Finkel and J.L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. In *Acta Informatica* 4 (1): 1–9, 1974.
[17] T. Ge and S. Zdonik. Handling Uncertain Data in Array Database Systems. In *ICDE*, 2008.
[18] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, P. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data.*SIGMOD*'08.
[19] A. Marathe and K. Salem. Query Processing Techniques for Arrays. In *VLDB Journal* 11: 68-91, 2002.
[20] M. Mitzenmacher, E. Upfal. *Probability & Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge U. Press, 2005.
[21] C. Re, N. Dalvi and D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In *ICDE*, 2007.
[22] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *ICDE*, 2007.
[23] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, R. Cheng. Database Support for Probabilistic Attributes and Tuples. In *ICDE*, 2008.
[24] M. Stonebraker et al. One size fits all? – Part 2. In *CIDR*, 2007.
[25] D. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein. BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models. In *VLDB*, 2008.
[26] http://db.csail.mit.edu/labdata/labdata.html.
[27] http://scidb.org/.

APPENDIX

A. Additional Examples of Array Data Correlation

Example A.1. *Quantization techniques in image or sound processing constrain some attribute from a continuous set of values (such as the real numbers) to a discrete set (such as the integers). It is a form of lossy compression. Once quantization has been done, the stored values become uncertain and the correlation is largely local.*

Example A.2. *A large network (e.g., the Internet) consists of nodes, each of which is a router. Consider the traffic at each node (e.g, the number of packets it needs to forward) at any point in time during a day. This traffic follows some distribution, and the distributions of neighboring nodes are correlated. All nodes form a two-dimensional array. A query may ask about routing decisions using knowledge of the network traffic distribution.*

B. Background on Bayesian Networks and Markov Random Fields

A *probabilistic graphical model* (PGM) is a diagrammatic representation of a probability distribution [5]. In a PGM, each node represents a random variable and *edges* express probabilistic relationships between these variables. There are two major classes of PGM's: Bayesian Networks (BN) and Markov Random Fields (MRF). BN's edges are *directed*, while MRF's edges are undirected. Directed graphs are useful for expressing causal relationships between random variables, whereas undirected graphs are better suited to expressing soft constraints between random variables. Figure 4(b) shows an example of BN. BN's joint probability density function can be written as a *product* of the individual density functions at node variables, conditioned on their parent variables.

Unlike BN, MRF uses undirected graphs to model random variables and their dependencies. Nodes in an MRF satisfies the "Markov property" which essentially says that all nodes are conditionally independent of the rest of the graph given their neighbors. The Hammersley-Clifford theorem [5] states that this is equivalent to the Gibbs property: the joint distribution of all nodes in the graph can be expressed as a product of multiple factors, each of which corresponds to a *clique* (i.e., a complete subgraph) and is a function of only random variables (nodes) within that clique.

C. Proofs of Theorems

Proof of Theorem 1: For $d = 1$, first of all, Lemma 1 is a simple property of a binary tree and is illustrated in Figure 5. The first cell from the left has label 0000, the second has 0001, and so on. Figure 5 shows the cases that CD = 0 to 3. Simply from the labels of the cells marked black we can determine its CD with its right neighbor.

Now consider the expected average CD. The label of a random cell comes from a random walk from the root to a leaf. Thus, *Pr*[zero trailing 1's] = ½, *Pr*[one trailing 1's] = ¼, etc. Let random variable A denote the average CD of the random q pairs. Then, from the linearity of expectation and Lemma 1, we have

$$E[A] = \sum_{i=1}^{h-1} i \frac{1}{2^{i+1}}$$

With some algebraic manipulation, which we omit, we get

$$E[A] = 1 + \frac{h+1}{2^h} \qquad (1)$$

We next compute the expected maximum CD. Let random variable X denote the maximum CD of q random pairs. Then we have,

$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] = \log q + \sum_{i=1+\log q}^{h-1} q\left(\frac{1}{2}\right)^i = \log q + 1 - \frac{q}{2^{h-1}} \qquad (2)$$

The first equality is due to the fact that X is nonnegative (intuitively, for i from 1 upwards, cumulatively, $Pr[X \geq i]$ is the probability that we add 1 to the expectation) [20]. $q(½)^i$ is the probability that any of the q pairs (hence the maximum) has CD at least i. This is effectively 1 for the first log$q$ terms, hence the second equality in the equation above.

Next we consider the case of $d = 2$. Labeling a 4-ary tree is similar. Each edge is now associated with a *2-bit label*, indicating the "left or right" decision for the two dimensions respectively. Thus, four children of a node have labels 00, 01, 10, and 11. To think about it another way, as a random walk is performed from the root to a leaf, we are in fact doing a random walk on *two binary trees* with the same height, one for each dimension. For a pair of neighboring cells along one dimension of the original 4-ary tree, they are next to each other in the binary tree of that dimension and are on the *same* leaf cell in the binary tree of the other dimension. From (1) we know that the expected average CD only depends on the height of the trees, but not q_1 or q_2. Thus, it is the same as in $d = 1$.

Let random variable Z denote the maximum CD; let random variables X and Y denote the maximum CD of the q_1 pairs along one dimension and that of the q_2 pairs along the other dimension, respectively. Thus, $Z = \max(X, Y)$. Similar to the reasoning in (2), we have

$$\begin{aligned} E[Z] &= \sum_{i=1}^{\infty} \Pr[Z \geq i] = \log q + \sum_{i=1+\log q}^{h-1} \left[1 - \left[1 - q_1\left(\frac{1}{2}\right)^i\right]\left[1 - q_2\left(\frac{1}{2}\right)^i\right]\right] \\ &= \log q + \sum_{i=1+\log q}^{h-1} \left[(q_1 + q_2)\left(\frac{1}{2}\right)^i - q_1 q_2 \left(\frac{1}{4}\right)^i\right] \\ &= \log q + (q_1 + q_2)\left[\frac{1}{q} - \left(\frac{1}{2}\right)^{h-1}\right] - \frac{q_1 q_2}{3}\left[\frac{1}{q^2} - \left(\frac{1}{4}\right)^{h-1}\right] \end{aligned}$$

This completes the proof of Theorem 1. □

Proof of Theorem 2: Consider each node C in a minimum cover. First we claim that if a minimum cover with subtraction does not include C, it must include a node (say, E) in the subtree of at least one of C's siblings (say, C'). This is because at least one of C's siblings covers a cell not in the target set of cells, otherwise C and its siblings all cover cells in the target and their *parent* node would be in the minimum cover, but not C. The minimum cover with subtraction must include E in order to subtract that cell. For example, in Figure 9, for node C in the minimum cover, the minimum cover with subtraction must contain a node (E) in the subtree of node C' (C's sibling). The same is true with node B.

Thus, to access the minimum cover with subtraction, one must access node C' (since it is the only way to reach node E in the top-down access of the minimum cover as discussed earlier). In other words, for each node in the minimum cover, when we use the minimum cover with subtraction instead, we must either access that node, or one of its siblings. Therefore, the two methods incur the same I/O cost. □

D. Additional Details of A*-trees

We now look at some details of an A*-tree, in particular, the representation of the joint distribution in each node and the layout of an A*-tree on disk.

D.1 Arbitrary Dimension Sizes

For ease of exposition, in Section 2 we assumed that each dimension has size 2^n (for some integer n). However, in reality, dimensions may have different sizes and they may not be a power of 2. We can partition the array in a similar fashion. Recall that the recursive partition of an array divides each dimension in half every time. We note two cases:

- We do the same even if a dimension is not a power of 2. When we have to divide a dimension of an odd size $2k + 1$. We simply divide it into pieces of size k and $k + 1$.
- When two dimensions do not have the same size, the "short" dimension must first reach size either 2 or 3 in the recursive partition procedure. At this point, we stop partitioning the short dimension but continue dividing the long dimension in halves, until the long dimension also reaches size 2 or 3. Now we have three combinations of block shape: 2 by 2, 2 by 3, and 3 by 3. As illustrated in Figure A.1, the first case is the same as the basic A*-tree; a final partition for the second case gives us a 1 by 2 and a 2 by 2 block; a final partition for the third case gives us 1 by 2, 1 by 3 and 2 by 2 blocks. Then the final joint distributions are on these blocks.

Note that each node of an A*-tree now keeps track of its bounding box (i.e., ranges that it covers at all dimensions). This will prove useful in Section 4 when query processing is discussed.

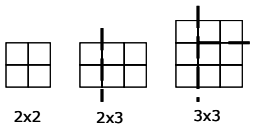


Fig A.1 Three combinations of final block shape and their partitions

D.2 Joint Distribution at a Node

In Section 2.1, we stated that a node encodes the joint distribution of its four children, relative to their average. We now elaborate on this and describe how to encode the joint distribution. Each node stands for the average of all cells in its subtree. Since each cell value is a random variable, so is each node value. Thus, we specify a joint distribution of X_1, X_2, X_3, and X_4, relative to a random variable Y (the average of X_1 to X_4), i.e., the joint distribution of the children (X_1 to X_4) given their parent's value (Y). But since X_4 is completely determined given Y, X_1, X_2, and X_3, we only need to specify the joint distribution of X_1, X_2, and X_3, relative to Y.

Because we need to represent X_i's relative to Y, usually this can be done either using a multiplicative factor or an additive term. Accordingly, the joint distribution relative to Y can be represented either (1) as a joint distribution of *multiplicative factors*, or (2) as a joint distribution of *additive offsets*. In the first method, we have $X_i = Y\ (1 + F_i)$, for $1 \le i \le 3$, where F_i is a *multiplicative factor*. We then simply encode the distributions of F_1, F_2, and F_3. In the second method, we have $X_i = Y + O_i$, where O_i is an *additive offset*, and we just encode the distributions of O_1, O_2, and O_3. We can use a histogram for both methods. Thus, they are similar and we only describe the first method.

Each of the F_i will have a range. There is a parameter k indicating the number of intervals for each F_i. Suppose there are r entries in the distribution table and each entry uses an l-bit number to represent the probability. Then the joint distribution takes $\frac{r \cdot (3\lceil \log k \rceil + l)}{8}$ bytes. Figure A.2 shows an example in which $k = 8$, $r = 8$ and $l = 4$ (i.e., probabilities are multiples of 1/16). Each F_i has 3 bits. The distribution table can be quite compact. Finally, recall that the *root* also holds the *distribution of the average value of the whole array*. This can either be a histogram or a well-known distribution (e.g., Gaussian).

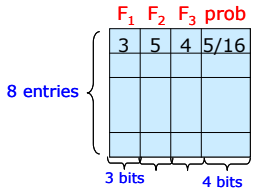


Fig. A.2 An example of a joint distribution table at a node

In general, obtaining a joint distribution is highly application specific. There are statistical methods to do this [5] and it is outside the scope of this paper. Having said that, we show a simplified example on how one might create an A*-tree.

Recall the sensor readings example in Section 1. Suppose the data in the array are temperatures at different locations in the space. However, the readings in the array are outdated and we have some uncertainty about what the current values are. The basic idea is that we "learn from the history". We examine logs for readings in the past, and figure out what correlation we can assume.

Time	X_1	X_2	X_3	X_4	Y	F_1	F_2	F_3
t*	72	73	71	74	72.5	-6.9	6.9	-20.7
t1	68	69	71	72	70	-28.6	-14.3	14.3
t2	69	71	69	71	70	-14.3	14.3	-14.3
t3	71	72	71	73	71.8	-11.1	2.8	-11.1
t4	77	75	76	78	76.5	6.5	-19.6	-6.5
t5	80	80	78	82	80	0	0	-25
t6	76	77	75	78	76.5	-6.5	6.5	-19.6
t7	72	73	71	74	72.5	-6.9	6.9	-20.7
t8	78	77	75	78	77	13	0	-26
t9	81	83	80	84	82	-12.2	12.2	-24.4

(a)

Time	F_1	F_2	F_3	d
t*	4	6	1	
t1	0	1	7	15
t2	2	7	2	4
t3	3	5	2	3
t4	6	0	3	10
t5	5	4	0	4
t6	4	6	1	0
t7	4	6	1	0
t8	7	4	0	6
t9	3	7	0	3

(b)

F_1	F_2	F_3	prob.
4	6	1	0.25
5	4	0	0.25
7	4	0	0.25
2	7	2	0.25

(c)

Fig A.3 History data (a), normalized data (b), and learned distribution (c)

We focus on four cells of the array. The highlighted first line in Figure A.3(a) indicates the data in the array. X_1 to X_4 are the values of four neighboring cells. Y and F_i's are computed as described earlier. The F_i values in Fig. A.3(a) have a scale factor of 10^{-3}. Our log contains readings in the past, at time t1 through t9. Our goal is to learn the correlation from the past. We first normalize the F_i's into interval numbers (0 to 7), as in Figure A.3(b). There are many ways to learn the distribution. For example, one can compute the L_1 distance between data entries in the past and the entry in the array (first line in Fig. A.3b) and find four entries that have the smallest distance. This is shown in the last column of Fig. A.3(b) as those four rows are highlighted. As a simplified illustration, we can use the F_i values in the four rows above them (i.e., the time instances *after* those entries that are closest to the values in the array) as entries in the joint distribution table and assign probability 0.25 to each (Figure

A.3c). Likewise, we can repeat this for nodes in the A*-tree at all levels.

Clearly, if we examine a constant number of log entries to learn each joint distribution, the overall cost of constructing this A*-tree is $O(n)$, where n is the number of non-empty cells of the array. This is because the total number of nodes of an A*-tree is also a constant factor of the number of non-empty cells of the corresponding array.

D.3 Layout on Disk

Typically, scientific data (e.g., astronomical images) is rarely updated. The data is mostly read-only. Our goal of managing an A*-tree on disk is thus to make it as compact as possible and read-optimized.

Succinct repersentation of k-ary trees is a well-studied problem in the literature. Various schemes have been proposed and analyzed (e.g., [4]). We choose to linearize an A*-tree in *level-order*: starting from the root level and descending one level at a time, nodes from left to right at each level are stored on disk in that order. This is one of the representations discussed in [4]. Figure A.4 shows an example in which we store the nodes in the numbered order bypassing the missing children. Note that as with any *positional tree*, we must record the information about which children are missing: we need that to determine cell locations.

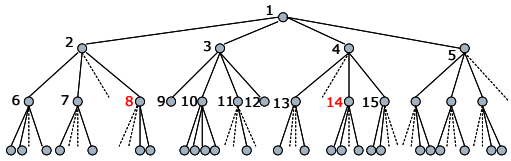


Fig. A.4 Illustrating a level-order storage of an A*-tree on disk

An advantage of storing nodes in level-order is that we only need to store the pointer to its first child at a node, as opposed to storing one pointer for each child. This is because other children must be stored immediately after the first child, likely in the same page. This makes the structure more compact. For example, in Figure A.4, node 3 only needs to store the pointer to its first child, node 9; other children immediately follow node 9.

E. Datasets and Setup of Experiments

We perform experiments on the following two datasets:

- **A real-world dataset**: We use the publicly available Intel Lab dataset [26]. It contains traces from a sensor network deployment which measures various physical attributes such as temperature, humidity, voltage of the sensors' batteries, etc. It uses the Berkeley Motes (sensor nodes) at several locations within the Intel Research Lab at Berkeley.
- **A synthetic dataset**: We also generate a dataset that is similar in nature to the Intel Lab dataset but can be arbitrary in size and sparsity.

We implement the A*-tree construction and query processing algorithms presented in the paper. All the experiments are carried out on a 1.6GHz AMD Turion 64 machine with 1GB physical memory.

ACKNOWLEDGEMENTS

We wish to thank the anonymous referees for several comments and suggestions that have improved the paper. Tingjian Ge was supported in part by the starup funding from the Department of Computer Science at the University of Kentucky. Stan Zdonik was supported in part by the NSF, under the grants IIS-0905553 and IIS-0916691.