APPENDIX

A. Computing $Pr^h(o \wedge C2)$

We compute $Pr^h(o \wedge C2)$ for all objects $o \in O$ together as a preprocessing step, which is executed only once for each dataset O.

In a possible world $pw \in C2$, all the objects are on the same line. Thus, two arbitrary objects $o_a, o_b \in O$ must occur on the line, and therefore this line can only be one of the $m_a \times m_b$ possible lines defined by an instance of o_a and an instance of o_b. To minimize the number of lines to check, we choose $a = \arg\min_i\{m_i\}$ and $b = \arg\min_{i \neq a}\{m_i\}$, i.e. o_a (and o_b) is the object with the smallest (and second smallest) number of instances.

For each line $\overline{s_a s_b}$ (i.e. under the condition that s_a and s_b occur), we check all the other objects $o_i \in O - \{o_a, o_b\}$ one by one, and meanwhile maintain a pool of key-value pairs.

Assume that we have already checked object instances $s_1 \in o_1$, $s_2 \in o_2$, $\ldots$, $s_{i-1} \in o_{i-1}$, then each key-value pair is composed of:

- Key $<s_\ell, s_r>$ that records the leftmost (and upmost) and the rightmost (and lowest) endpoints of $s_1, s_2, \ldots, s_{i-1}$. Note that s_ℓ and s_r are instances.
- Value $Pr\{\bigwedge_{t=1}^{i-1}\{s_t \wedge onLine(\overline{s_a s_b}, s_t)\} \mid s_a, s_b\}$, where $onLine(\overline{s_a s_b}, s_t)$ refers to the event that s_t lies on line $\overline{s_a s_b}$.

Initially, there is only one pair $(<s_a, s_b>, 1)$ or $(<s_b, s_a>, 1)$ in the pool. We perform the following operations when checking object o_i:

- As long as object o_i is found to have no instance on $\overline{s_a s_b}$, we know that it is impossible for all the objects to be on $\overline{s_a s_b}$, and stop checking $\overline{s_a s_b}$.
- Otherwise, for each $s_i \in o_i$ on $\overline{s_a s_b}$, we expand each pair in the pool by (1) updating the key if s_i becomes a new endpoint, and (2) multiplying the old value by $Pr(s_i)$ to incorporate the event $onLine(\overline{s_a s_b}, s_i)$ into the probability.

In fact, we can merge those pairs with the same key $<s_\ell, s_r>$ after checking each object, by summing up their values, so as to keep the pool size small. This modification, in effect, changes the meaning of the value of a key-value pair to $Pr\{\bigwedge_{t=1}^{i-1}\{onLine(\overline{s_a s_b}, s_t)\} \wedge s_\ell, s_r$ are endpoints $\mid s_a, s_b\}$.

Therefore, after all the objects $o_i \in O - \{o_a, o_b\}$ are checked to have instances on $\overline{s_a s_b}$, each pair with key $<s_\ell, s_r>$ has value equal to $Pr\{$all objects are collinear $\wedge\ s_\ell, s_r$ are endpoints $\mid s_a, s_b\}$, or equivalently, $Pr\{s_\ell, s_r$ are endpoints $\wedge\ C2 \mid s_a, s_b\}$.

Note that when s_ℓ, s_r are endpoints, objects o_ℓ and o_r are on the convex hull. Therefore, given the pairs in the pools obtained after all lines are checked, we can compute $Pr^h(o_j \wedge C2)$ by the following formula:

$$Pr^h(o_j \wedge C2) = \left[\sum_{s_a \in o_a}\sum_{s_b \in o_b}\left(\sum_{\ell=j} + \sum_{r=j}\right)\right] Pr(s_a) \cdot Pr(s_b) \cdot Pr\{s_\ell, s_r \text{ are endpoints } \wedge C2 \mid s_a, s_b\}. \quad (23)$$

The above equation is formulated according to the additivity of probability, since the events $\{s_a, s_b$ occur $\wedge$ all objects are on $\overline{s_a s_b}$ with endpoints $s_\ell, s_r\}$ are disjoint.

The algorithm of computing $Pr^h(o \wedge C2)$ may take time exponential to the data size in the worst case (e.g. when all instances of all objects are on the same line). Fortunately, for the majority of the real world data, it is very unlikely that all objects are collinear. Line checking usually stops after examining a few objects $o_i \in O - \{o_a, o_b\}$.

Furthermore, in Section IV-B, we will present an *object-level four-corner pruning* technique, and show that as long as the method prunes some object, $Pr^h(o \wedge C2)$ are guaranteed to be 0 for all $o \in O$, and the preprocessing step described here does not even need to be executed.

B. R-Tree Traversal Operations When Computing $p(\widehat{s_j s_i s_k})$

Let us denote the MBR of node $\mathcal{N}$ as $\mathcal{N}.M$. Following are several important details during the traversal of T_O:

- We can prune a node $\mathcal{N}$ if we find $ccw(\overrightarrow{s_j s_i}, \mathcal{N}.M) \wedge ccw(\overrightarrow{s_i s_k}, \mathcal{N}.M)$, since $\forall o \in \mathcal{N}$, o is $V(o)$-full.
- If we traverse to a node $\mathcal{N}$ satisfying $cw(\overrightarrow{s_j s_i}, \mathcal{N}.M) \vee cw(\overrightarrow{s_i s_k}, \mathcal{N}.M)$, then $\forall o \in \mathcal{N}$, o is $V(o)$-empty. Thus, we terminate and return $p(\widehat{s_j s_i s_k}) = 0$.
- When we reach a leaf node, we filter out entries $o_i.M$, $o_j.M$ and $o_k.M$, since we only consider $o_t \in O - \{o_i, o_j, o_k\}$ in Equations (8) and (9).

We traverse aR_{o_t} to compute the result $\sum_{s_t \in V(o_t)} Pr(s_t)$ (or $|V(o_t)|$), which is initialized to 0. Following are several important details when traversing aR_{o_t}:

- If we traverse to a node $\mathcal{N}$ satisfying $ccw(\overrightarrow{s_j s_i}, \mathcal{N}.M) \wedge ccw(\overrightarrow{s_i s_k}, \mathcal{N}.M)$, then $\forall s_t \in \mathcal{N}$, $s_t \in V(o_t)$. Thus, we directly add the aggregate value of $\mathcal{N}$ to the result.
- If we traverse to a node $\mathcal{N}$ satisfying $cw(\overrightarrow{s_j s_i}, \mathcal{N}.M) \vee cw(\overrightarrow{s_i s_k}, \mathcal{N}.M)$, we prune $\mathcal{N}$ since $\forall s_t \in \mathcal{N}$, $s_t \in \overline{V(o_t)}$.
- When we reach a leaf node and process instance s_t, we add $Pr(s_t)$ (or 1 when all instances of any object are equally likely to occur) to the result if we find $s_t \in V(o_t)$ after checking Equation (7).

Note that an object $o_t \in C$ may still be $V(o_t)$-empty since we only check $o_t.M$ when traversing T_O, and we terminate and return $p(\widehat{s_j s_i s_k}) = 0$ once we find such an object o_t after traversing aR_{o_t}.

C. Proof of Theorem 2

Proof: Given an object o, its MBR $o.M$ determines four regions as shown in Figure 15. Suppose that there exists four objects o_1, o_2, o_3 and o_4, such that $o_1.M$, $o_2.M$, $o_3.M$ and $o_4.M$ are totally contained in Regions I, II, III, IV, respectively. Then, in any possible world pw, s_1, s_2, s_3 and s_4 are totally contained in Regions I, II, III, IV, respectively.

We now prove that the convex hull in pw is a polygon that contains $o.M$, so that any instance $s \in o$ is not on the convex hull. Due to the arbitrary of pw, we would have $Pr^h(o) = 0$.

Let us define $O' = \{o_1, o_2, o_3, o_4, o\}$. Since $O' \subseteq O$, the convex hull of O' in pw is a polygon that is contained in the

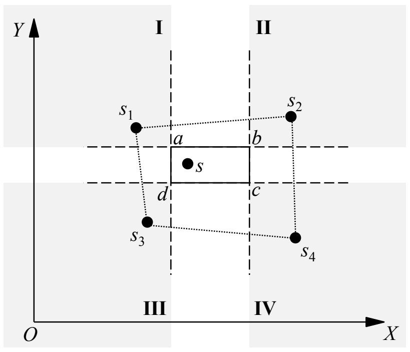


Fig. 15. Proof of Theorem 2

polygon defined by the convex hull of O in pw. Therefore, it is sufficient if we can prove that the polygon defined by the convex hull of O' in pw contains $o.M$.

We now prove that the convex hull of O' in pw, i.e. polygon $s_1s_2s_3s_4$ in Figure 15, contains $o.M$, i.e. rectangle $abcd$ in Figure 15. Without loss of generality, we only need to prove that polygon edge s_1s_2 is above $o.M$, or equivalently, above rectangle edge ab. This holds because both $s_1.y$ and $s_2.y$ are at least y_2, which accomplishes the proof.

The case of instance-level four-corner pruning can be similarly proved. ■

D. Containment query in Four-Corner Pruning

Our four-corner pruning operation is based on four range queries on the object R-tree T_O. We call such range queries the *containment queries*, which return whether there exists an object whose MBR is totally contained in the query region defined by query object o_q or query instance s_q.

We now illustrate how a *containment query* with Region I $= \{(x,y) \in \mathbb{R}^2 | x \leq x_1 \wedge y \geq y_2\}$ works. The most important details are listed below:

- During the traversal of T_O, if a node $\mathcal{N}$ has MBR $\mathcal{N}.M$ totally contained in region $R_1 = \{(x,y) \in \mathbb{R}^2 | x < x_1 \wedge y > y_2\}$, then $\forall o \in \mathcal{N}$, we have $o \neq o_q$ and $o.M$ is totally contained in $R_1 \subset$ Region I. Thus, we terminate the *containment query* immediately and return a positive answer.
- When we reach a leaf node and check whether entry $o.M$ is totally contained in Region I, we need to filter out object o_q.

As long as one of the four *containment queries* gives a negative answer, we can return a negative answer immediately without evaluating the remaining queries.

E. Algorithm for Four-Corner Upper Bounding

In order to make $UB^h(s)$ tight, for each Region $t \in$ {I, II, III, IV}, we choose objects o_t to maximize $\sum_{s_t \in \text{Region } t} Pr(s_t)$ (or n_t/m_t when all the instances of any object are equally likely to occur), which is achieved by the following three steps:

1) Perform a range query on the object R-tree T_O with Region t as the query window, to find the set of (intersecting) objects $C_t(s) = \{o_r \in O - \{o\} | o_r.M \cap \text{Region } t \neq \emptyset\}$;
2) For each object $o_r \in C_t(s)$, compute $\sum_{s_r \in \text{Region } t} Pr(s_r)$ (or n_r/m_r) by performing a range query on the instance aR-tree aR_{o_r} with Region t as the query window;
3) Return the maximum of the computed values.

Given query instance $s = (x,y) \in o$, we now illustrate how a *range query* with Region I= $\{p \in \mathbb{R}^2 | p.x \leq x \wedge p.y \geq y\}$ works (Step 1). The most important details are listed below:

- During the traversal of T_O, if a node $\mathcal{N}$ has MBR $\mathcal{N}.M$ totally contained in region $R_1 = \{p \in \mathbb{R}^2 | p.x < x \wedge p.y > y\}$, then $\forall o_r \in \mathcal{N}$, we have $o_r \neq o$ and o_r is totally contained in $R_1 \subset$ Region I. Therefore, we return $\max_t\{\sum_{s_t \in \text{Region } t} Pr(s_t)\} = 1$ (or $\max_t\{n_t/m_t\} = 1$) immediately without going to the 2nd and 3rd steps.
- Suppose we traverse to a node $\mathcal{N}$ whose MBR $\mathcal{N}.M$ has lower-left and upper-right corners (x_1, y_1) and (x_2, y_2), repectively. If $x_1 > x$ or $y_2 < y$, we prune $\mathcal{N}$ since it is totally outside of Region I.
- When we reach a leaf node, we need to filter out the object o of query instance s, as well as those objects already picked when processing other regions.

The last operation is necessary because we require o_{I}, o_{II}, o_{III} and o_{IV} to be different in Equation (15).

After obtaining $C_t(s)$ in the first step, we perform a range query on aR_{o_r} for each $o_r \in C_t(s)$ to compute $\sum_{s_r \in \text{Region } t} Pr(s_r)$ or n_r/m_r. This operation is similar to that of computing Equations (10) and (11) in Section IV-A (where $V(o)$ is involved instead of $C_t(s)$), except that only coordinate comparison is required here instead of *ccw indicator* evaluation.

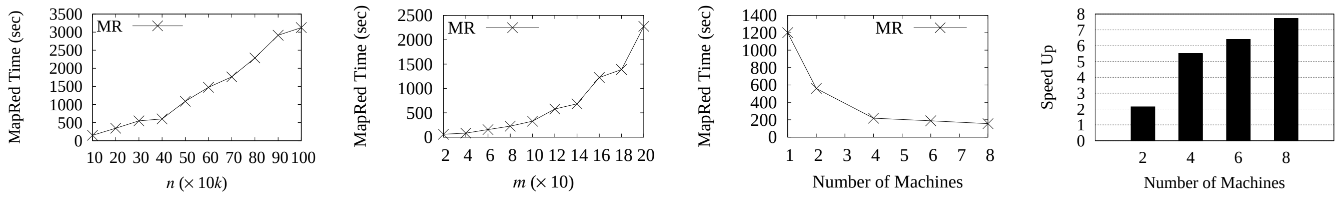
F. MapReduce Algorithm

Query evaluation on uncertain data is usually much more expensive than if the data is deterministic. This is because of the problem of "possible world explosion" caused by the possible world semantics, which is popularly used in modeling data uncertainty.

For example, [19] shows that the data complexity of some SQL queries on uncertain databases is *#P-complete*, which implies that these queries do not admit any efficient evaluation methods. Our problem is of no exception: while it takes only $O(N \log N)$ time to find the convex hull of a 2D point set P ($|P| = N$), the time complexity becomes $O(N^3)$ when the data is uncertain.

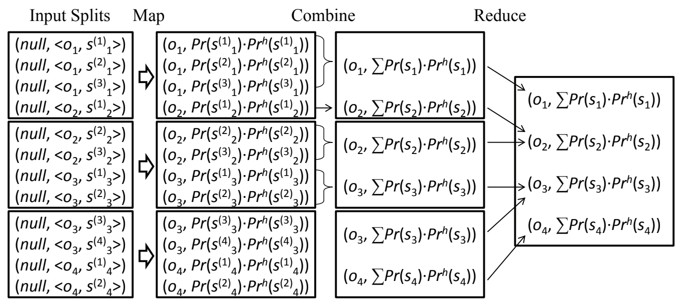
Fortunately, our $\mathcal{CI}$ algorithm can be easily parallelized: while the evaluation of $Pr^h(s_i)$ is inseparable due to the common structure A_{s_i} used in the process, we can evaluate $Pr^h(s)$ for different instances s in parallel, using MapReduce. As far as we know, we are the first to use MapReduce to attack the computational overhead brought by data uncertainty. Figure 16 illustrates our MapReduce algorithm for computing $PCH_\alpha(O)$ given four objects o_1–o_4.

Initially, we perform the preprocessing step on the client machine to obtain the list of non-pruned instances ordered



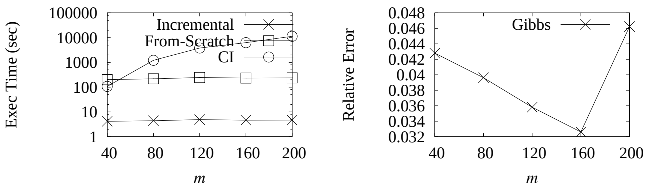
(a) Exec. Time w.r.t. n (b) Exec. Time w.r.t. m (c) Exec. Time w.r.t. Cluster Size (d) Speed up w.r.t. Cluster Size

Fig. 17. Experimental Results on MapReduce



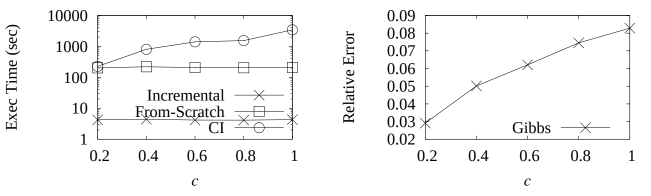
Input Splits
Map
Combine
Reduce
$(null, <o_1, s^{(1)}_1>)$
$(null, <o_1, s^{(2)}_1>)$
$(null, <o_1, s^{(3)}_1>)$
$(null, <o_2, s^{(1)}_2>)$
$(null, <o_2, s^{(2)}_2>)$
$(null, <o_2, s^{(3)}_2>)$
$(null, <o_3, s^{(1)}_3>)$
$(null, <o_3, s^{(2)}_3>)$
$(null, <o_3, s^{(3)}_3>)$
$(null, <o_3, s^{(4)}_3>)$
$(null, <o_4, s^{(1)}_4>)$
$(null, <o_4, s^{(2)}_4>)$
$(o_1, Pr(s^{(1)}_1)\cdot Pr^h(s^{(1)}_1))$
$(o_1, Pr(s^{(2)}_1)\cdot Pr^h(s^{(2)}_1))$
$(o_1, Pr(s^{(3)}_1)\cdot Pr^h(s^{(3)}_1))$
$(o_2, Pr(s^{(1)}_2)\cdot Pr^h(s^{(1)}_2))$
$(o_2, Pr(s^{(2)}_2)\cdot Pr^h(s^{(2)}_2))$
$(o_2, Pr(s^{(3)}_2)\cdot Pr^h(s^{(3)}_2))$
$(o_3, Pr(s^{(1)}_3)\cdot Pr^h(s^{(1)}_3))$
$(o_3, Pr(s^{(2)}_3)\cdot Pr^h(s^{(2)}_3))$
$(o_3, Pr(s^{(3)}_3)\cdot Pr^h(s^{(3)}_3))$
$(o_3, Pr(s^{(4)}_3)\cdot Pr^h(s^{(4)}_3))$
$(o_4, Pr(s^{(1)}_4)\cdot Pr^h(s^{(1)}_4))$
$(o_4, Pr(s^{(2)}_4)\cdot Pr^h(s^{(2)}_4))$
$(o_1, \sum Pr(s_1)\cdot Pr^h(s_1))$
$(o_2, \sum Pr(s_2)\cdot Pr^h(s_2))$
$(o_2, \sum Pr(s_2)\cdot Pr^h(s_2))$
$(o_3, \sum Pr(s_3)\cdot Pr^h(s_3))$
$(o_3, \sum Pr(s_3)\cdot Pr^h(s_3))$
$(o_4, \sum Pr(s_4)\cdot Pr^h(s_4))$
$(o_1, \sum Pr(s_1)\cdot Pr^h(s_1))$
$(o_2, \sum Pr(s_2)\cdot Pr^h(s_2))$
$(o_3, \sum Pr(s_3)\cdot Pr^h(s_3))$
$(o_4, \sum Pr(s_4)\cdot Pr^h(s_4))$

Fig. 16. MapReduce Algorithm Illustration



(a) Exec. Time (b) Relative Error

Fig. 18. Effect of m on Gibbs Sampling



(a) Exec. Time (b) Relative Error

Fig. 19. Effect of c on Gibbs Sampling

by object ID. Then, we use the "NLineInputFormat" class of Hadoop to partition the instance list into f splits that have the same number of instances, each of which is submitted to a map task. For each input instance s_i, a mapper computes $Pr^h(s_i)$ using the batch evaluation technique, and outputs a key-value pair $(o_i, Pr(s_i)Pr^h(s_i))$. According to Equation (12), $Pr^h(o_i)$ is obtained by summing up the values of all mapper output records with key o_i. Therefore, given key o_i, a combiner simply adds up all the values on the local slave machine, so as to reduce the number of records to send (to the reducer). Finally, a single reducer adds up all the received values for each object (and key) o_i, the result of which is exactly $Pr^h(o_i)$. The final records $(o_i, Pr^h(o_i))$ output by our MapReduce algorithm is ordered by object ID due to the sorting phase before the reduce phase.

Note that the mappers require the data information (e.g. for getting $Pr(s_i)$ and for bulk-loading R-trees) and the pruning/upper bouding results. These information are stored on Hadoop Distributed File System (HDFS) after preprocessing, and loaded into memory whenever a mapper processes the first record in a split. The number of splits f influences the running time of our MapReduce algorithm: if f is too small, good load-balancing may not be achieved; if f is too large, the overhead of loading data from HDFS becomes non-negligible (f splits imply f map tasks, which in turn imply f times of data loading from HDFS). We have done extensive experiments on the choice of f, and find that reasonable performance can usually be achieved when f is set to be 5 times the number of processors in the cluster.

Experimental Results. Our MapReduce algorithm was run on a cluster of 8 machines, each with two 2.8GHz Intel CPU and 2GB memory. The MapReduce framework we use is Hadoop 0.20.2.

To test the scalability of our MapReduce algorithm, we repeated the first two sets of experiments in Table II, using the MapReduce algorithm on a cluster of 8 machines. The results of running time are shown in Figures 17(a) and (b), which show significant performance improvement compared with the results in Figures 10–11(a).

To study the influence of cluster size on the running time of our MapReduce algorithm, we generated 10 datasets with configuration $(n, m, c) = (10^4, 20, 0.2)$, and compute $PCH_\varepsilon(O)$ of those datasets on Hadoop, which is configured to work on clusters of 2, 4, 6, and 8 machines. All the results are reported based on the average of the 10 datasets.

Note that given a cluster of i machines, one machine serves as the master, and the other $i-1$ machines are slaves. Since each machine has two processors, at most $2(i-1)$ map tasks are active at each time instant. We set the number of input splits $f = 10(i-1)$ in all the experiments. Figure 17(c) shows the running time of our MapReduce algorithm except for the first result, which is the running time of $\mathcal{CI}$ on a single machine. Figure 17(d) shows the speed up of our MapReduce algorithm w.r.t. $\mathcal{CI}$ for evaluating $PCH_\varepsilon(O)$, where almost 8 times speed up is achieved when there are 8 machines in the cluster.

G. Additional Experimental Results on Gibbs Sampling

Figure 18(a) (and (b)) shows the running time (and accuracy) of our algorithms when $(n, c, k) = (1000, 0.2, 5 \times 10^5)$ and m varies. Figure 18(a) shows that *Incremental* always finishes in several seconds, and is orders of magnitude faster than *From-Scratch* and $\mathcal{CI}$. Figure 18(b) shows that the accuracy of Gibbs sampling decreases as m increases.

Figure 19(a) (and (b)) shows the running time (and accuracy) of our algorithms when $(n, m, k) = (1000, 40, 5 \times 10^5)$ and c varies. Figure 19(a) shows that *Incremental* always finishes in several seconds, and is orders of magnitude faster than *From-Scratch* and $\mathcal{CI}$. Figure 19(b) shows that the accuracy of Gibbs sampling decreases as c increases.

Note that in Figure 14(a), *From-Sketch* is even more expensive than the exact algorithm $\mathcal{CI}$. However, as m and c becomes larger, $\mathcal{CI}$ becomes much more expensive than *From-Sketch*, as shown in Figure 18(a) and Figure 19(a). Since *Incremental* always stops in several seconds and gives accurate probability estimations (the relative error is much smaller than 5%), it is able to support real-time applications such as animal tracking and analysis. On the other hand, *From-Sketch* is a baseline for comparison that is much more expensive than *Incremental* but returns the same result as *Incremental*, and thus we should never use it.