# Confidence-Aware Join Algorithms*

Parag Agrawal, Jennifer Widom
Stanford University
{paraga,widom}@cs.stanford.edu

***Abstract*— In uncertain and probabilistic databases, *confidence* values (or *probabilities*) are associated with each data item. Confidence values are assigned to query results based on combining confidences from the input data. Users may wish to apply a threshold on result confidence values, ask for the "top-$k$" results by confidence, or obtain results sorted by confidence. Efficient algorithms for these types of queries can be devised by exploiting properties of the input data and the combining functions for result confidences. Previous algorithms for these problems assumed sufficient memory was available for processing. In this paper, we address the problem of processing all three types of queries when sufficient memory is not available, minimizing retrieval cost. We present algorithms, theoretical guarantees, and experimental evaluation.**

## I. Introduction

In uncertain and probabilistic databases, *confidence* values (or *probabilities*) are associated with data items [9], [5], [1], [10], [19]. For example, a tuple $\langle \texttt{ssn}, \texttt{zipcode} \rangle$ might have a confidence associated with it denoting the likelihood of the person identified by `ssn` living in `zipcode`. Results of queries typically also have confidence values associated with them. For example, a join of $\langle \texttt{ssn}, \texttt{zipcode} \rangle$ with tuple $\langle \texttt{ssn}, \texttt{name} \rangle$ (also having a confidence value) produces $\langle \texttt{name}, \texttt{zipcode} \rangle$, with an associated confidence based on combining the confidences of the joining tuples. When result data has confidence values, the following query types become important:

- **Threshold**: Return only result tuples with confidence above a threshold $\tau$.
- **Top-$k$**: Return $k$ result tuples with the highest confidence values.
- **Sorted**: Return result tuples sorted by confidence.
- **Sorted-Threshold**: Return result tuples with confidence above a threshold $\tau$, sorted by confidence.

In our experience with Trio [21], a significant fraction of queries over uncertain data belong to these four types. We consider these query types applied to joins of uncertain (probabilistic) relations. We assume that a result tuple's confidence is computed from the confidence values of the joining tuples using a *combining function*. Combining functions are typically monotonic: combining higher (respectively lower) values produces a higher (respectively lower) result. For instance, in the probabilistic setting with independent joining tuples, the combining function is multiplication, which is monotonic for confidences in $[0, 1]$. We consider a setting where stored relations provide efficient sorted access by confidence, e.g., through a primary index. We exploit monotonicity of the combining function and sorted access on the relations to devise efficient algorithms for all four query types above.

An alternative approach could be not to devise special new algorithms, but rather simply treat confidence values as a regular primary-indexed column. Then, join queries can specify the output confidence as an explicit arithmetic expression wherever it is used. For instance, a threshold condition could be a filter in the `Where` clause, and a sort could use an `Order By` on the expression. Then we hand our query to a traditional optimizer. The problem with this approach is that traditional optimizers do not take advantage of the monotonicity in the combining function, which can yield more efficient execution techniques. This observation has been made before in [14], where the techniques of [8] are incorporated into a "rank-aware" join operator. However, that operator assumes sufficient memory is available to perform the join. In this paper, we provide efficient algorithms for the case where memory is insufficient. Note that threshold and sorted queries in particular may operate over a large fraction of a large uncertain database.

We cast our work in the context of uncertain databases with confidence values. However, like [8], [14], the techniques are generic: they apply to any setting where ranked data items are combined with a monotonic function.

The outline and contributions of this paper are as follows:

- In Section II we formalize our problem and environment: processing joins over uncertain data with confidences, in which result confidences influence query processing and data sets are large so memory may be a limitation.
- In Section III we provide efficient algorithms for all four query types introduced above: *Threshold*, *Top-k*, *Sorted*, and *Sorted-Threshold*. Our algorithms are suitable for processing stand-alone join queries, and they can be used for non-blocking operators within a larger query plan.
- Section IV presents efficiency guarantees for our algorithms. For each query type and problem instance, we present our guarantees with respect to the most efficient correct algorithm on that instance, that uses the same primitive operations and memory as ours.
- Section V describes our performance experiments. We use a large synthetic data set to evaluate performance of our algorithms against the theoretical lower-bounds, and to see how they are affected by different inputs and parameters. Our results show performance very close to the lower-bounds. In addition, we show empirically that we can deliver results sorted by confidence with cost proportionate to the number of result tuples delivered.

*This work was supported by the National Science Foundation under grants IIS-0324431 and IIS-0414762, by grants from the Boeing and Hewlett-Packard Corporations, and by a Stanford Graduate Fellowship from Cisco Systems.

Related work is covered in Section VI and we conclude in Section VII.

## II. Foundations

Consider two relations $R$ and $S$. Generically, a join operation $R \bowtie_\theta S$ on them (where $\theta$ is the join condition) can be viewed as an exploration of the cross-product of $R$ and $S$. For example, if:

$$R = \{\langle \texttt{ssn1}, \texttt{94305}\rangle, \langle \texttt{ssn2}, \texttt{10025}\rangle, \langle \texttt{ssn3}, \texttt{94041}\rangle\}$$
$$S = \{\langle \texttt{ssn3}, \texttt{Bob}\rangle, \langle \texttt{ssn4}, \texttt{Jane}\rangle\}$$

then six points in the cross-product need to be considered to evaluate $R \bowtie_\theta S$. All join methods effectively explore this two-dimensional space, though often incorporating reorganization of tuples or indexing to enable rapid pruning. In our setting, we do not assume any indexes on the joining attributes. Instead, our pruning is based on input and result confidences, and the query type under evaluation. Overall, we view our algorithms as one option to be considered by a query optimizer.

Consider the join of $R$ and $S$ where neither relation fits in memory. To evaluate the join condition on a pair of tuples, both must simultaneously be in memory. Clearly, to explore the entire cross-product, some tuples must be read into memory multiple times. A nested-loop join would allocate some memory $M$ to load a part of the "outer" relation, say $S$, while it scans through the "inner" relation $R$. This nested-loop join can be visualized as illustrated in Figure 1. A vertical double-arrow represents a load operation on the outer (in this case $S$); and an arrow perpendicular to it represents a scan on the inner (in this case $R$); joining with those tuples from the outer that are in memory. The rectangle thus formed is the part of the cross-product explored in one "load and scan". The breadth (smaller dimension) of a rectangle is limited by memory $M$. The cost of the entire join is proportionate to the sum of the length and breadth of the rectangles. If $S$ were the inner, then the rectangles would be vertical.

Now let us add confidences. Let $c(t)$ denote the confidence of a tuple $t$. Also let $f$ be the combining function: the confidence of the join of tuples $r$ and $s$ is $f(r, s)$. A combining function $f$ is monotonic iff $f(r_i, s_j) \geq f(r_k, s_l)$ whenever $c(r_i) \geq c(r_k)$ and $c(s_j) \geq c(s_l)$. Recall, we are assuming that relations $R$ and $S$ can be retrieved efficiently sorted by descending confidence. Thus, we assume $c(r_i) \geq c(r_j)$ and $c(s_i) \geq c(s_j)$ whenever $i \leq j$, where indices represent order of tuples. In Figure 2, the confidence of tuples of $R$ decreases as we move right and correspondingly for $S$ it decreases as we move up.

A point in the two dimensional space represents a (potential) result tuple whose confidence combines those on its dimensions. Suppose we have a threshold value $\tau$ for the result confidence. Then the part of the cross-product that can contribute to the result is guaranteed to be a stair-like area as illustrated in Figure 2. Subsequently, we refer to this area as the *shaded region*. Intuitively, the value of the combining function for a point $p$ is a lower bound for all points with lower $x$ and $y$ positions, and hence all such points belong to

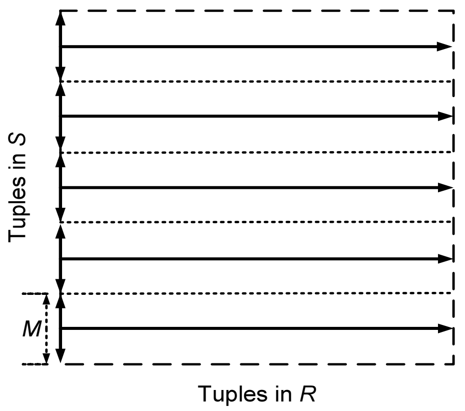

**Fig. 1:** *Nested-Loop Join*

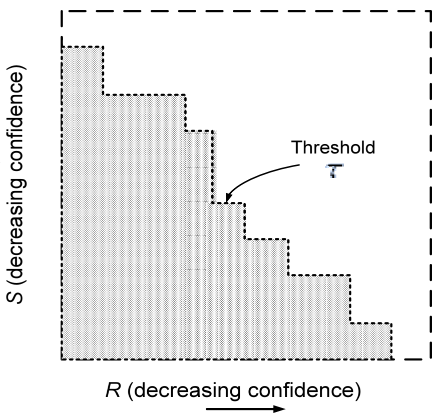

**Fig. 2:** *Shaded Region*

the shaded region if $p$ does. Formally, the following holds: $f(r_i, s_j) \geq \tau$ implies $f(r_k, s_l) \geq \tau$ for all $k \leq i$ and $l \leq j$, by monotonicity of $f$.

The "stair" observation allows us to answer queries without exploring the entire space. For instance, a query with threshold $\tau$ needs to explore only the shaded region in Figure 2, rather than the entire cross-product. The other query types we consider can also benefit from this observation, and the contiguous shape of this region allows efficient exploration as we shall discuss.

## III. Algorithms

### A. Basics

In this section, we set up the basics for the description of the algorithms. Recall we assume that relations can be retrieved through sorted access by confidence. More precisely, we assume that we can ask for tuples in a relation starting at an offset in sorted order. We also assume a monotonic combining function $f$ that is used to compute confidence values of result tuples using confidence values on the joining tuples. We consider a cost metric proportionate to data retrieval cost. This metric closely resembles disk read cost, since our algorithms all retrieve data in large sequential chunks. Some data items may be retrieved more than once and are counted as many times as they are retrieved. (We manage all use of memory as part of our algorithms.)

We use the following symbols in our description:

- $R$ and $S$ are the relations to be joined, with $R$ usually the inner and $S$ the outer.
- $o_r$, $o_s$, $o_{r_1}$, $o_{r_2}$, $o_{s_1}$, and $o_{s_2}$ are offsets in the sorted relations, in some unit of memory.
- $M$ and $L$ are memory sizes in the same unit as the offsets above.

Our algorithms use the following operations:

- **load**($S$, $o_{s_1}$, $o_{s_2}$): Loads into memory tuples from relation $S$ (outer) starting at offset $o_{s_1}$ and ending at $o_{s_2}$. (It may construct a hash table on the joining attributes to enable efficient look-ups in memory when $R$ is scanned, if the join condition can benefit, but doing so does not affect retrieval cost). The cost of a load is $o_{s_2} - o_{s_1}$.
- **scan**($R$, $o_{r_1}$, $o_{r_2}$): Scans relation $R$ (inner) starting at offset $o_{r_1}$ and ending at $o_{r_2}$. While scanning, the join condition is evaluated on each scanned tuple of $R$ and all tuples of $S$ residing in memory (possibly using the hash table if the join condition allows it). The cost of a scan is $o_{r_2} - o_{r_1}$.
- **conf**($R$, $S$, $o_r$, $o_s$): Returns the result of the combining function $f$ on the tuple from $R$ at offset $o_r$ and tuple of $S$ at offset $o_s$. A negative value (flag) is returned if the offset for either relation exceeds the size of the relation. This function is always called such that the data required is in memory.
- **explore**($R$, $S$, $o_{r_1}$, $o_{r_2}$, $o_{s_1}$, $o_{s_2}$): Combines *load*($S$, $o_{s_1}$, $o_{s_2}$) followed by *scan*($R$, $o_{r_1}$, $o_{r_2}$). Each *explore* step covers a rectangle of the two-dimensional space as shown in Figure 3. The cost for the exploration of this rectangle is the sum of the length corresponding to the *scan* and the breadth corresponding to the *load*: $(o_{r_2} - o_{r_1}) + (o_{s_2} - o_{s_1})$.

In *load*, *scan*, and *explore*, sometimes $o_{s_2}$ and $o_{r_2}$ are not constants, but are found by checking a condition while loading or scanning. In our pseudo-code, these end-conditions are specified alongside the function call.

We consider the setting where the total memory available is limited: specifically there is only enough memory to *load* size $M$ of a relation, plus a small amount of additional memory for scanning and maintaining state for the algorithms. Note that since breadth corresponds to *load*, *explore* rectangle can have breadth no more than $M$.

## B. Running Example

We will use a running example to illustrate the discussion of the algorithms. For this example, both relations $R$ and $S$ are of size $5M$, and the combining function is multiplication. Their cross-product is visualized in Figure 4. The confidences of tuples with offsets that are multiples of $M$, namely $0, M, 2M, 3M$, and $4M$ are shown. The two-dimensional space is divided into 25 blocks named $a, b, ..., y$ that are of length and breadth $M$. The confidence of the (potential) result tuple in the lower-left corner of each block can be computed using the given confidences on the axes.

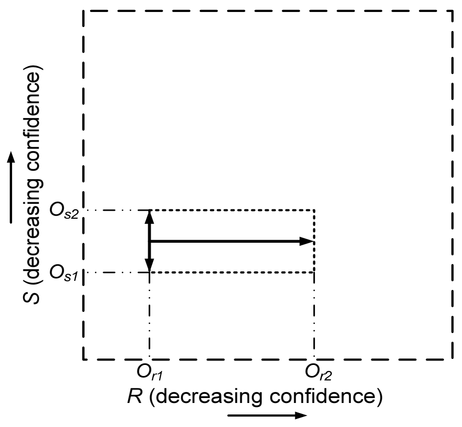

**Fig. 3:** *One* explore *step*

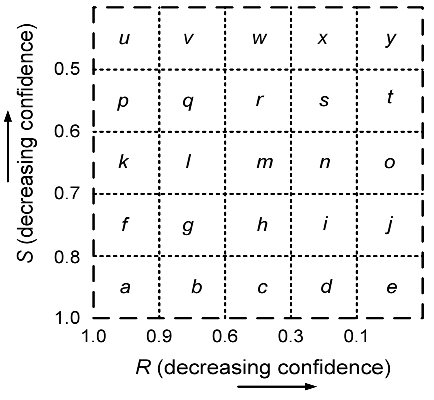

**Fig. 4:** *Running Example*

## C. Threshold

A *Threshold* query returns all result tuples whose confidence value is above a specified threshold $\tau$. We describe algorithms to efficiently compute results of *Threshold* queries over joins of two relations. As discussed in Section II, to answer such queries, we only need to explore a stair-like region in the cross-product. This exploration can be performed like a nested-loop join, modified not to explore regions that cannot contribute results with confidence above the threshold. The exploration is detailed in Algorithm 1 and illustrated in Figure 5. We call this algorithm *Threshold1*.

**Algorithm 1** Threshold1

```
$i \leftarrow 0$
while 1 do
    if $conf(R, S, 0, M \cdot i) \leq \tau$ then
        $break$
    $explore(R, S, 0, o_r, M \cdot i, \min(M \cdot (i+1), o_s))$
                    $\{o_s \leftarrow \min_p conf(R, S, 0, p) \leq \tau\}$
                    $\{o_r \leftarrow \min_p conf(R, S, p, M \cdot i) \leq \tau\}$
    $i \leftarrow i + 1$
```

In the example of Figure 4, for $\tau = 0.55$, the blocks would be explored in the following order: $a, b, c, f, g, k, l, p$. Blocks $c, g, l, p$ are explored only partially: the load and scan within *explore* operations also use threshold $\tau$ to terminate, as seen in the specification of $o_s$ and $o_r$ in Algorithm 1. For instance,

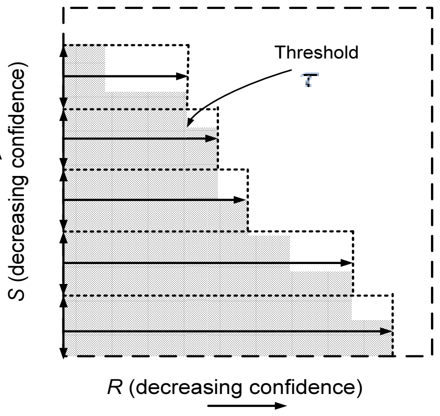

**Fig. 5:** *Algorithm* Threshold1

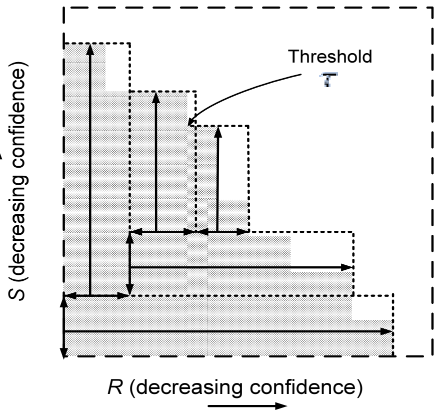

**Fig. 6:** *Algorithm* Threshold2

in the example the load on block $p$ only loads tuples in $S$ with confidence $> 0.55$, and scans only tuples in $R$ with confidence $> 0.55/0.6$. The *explore* operation emits result tuples as it finds them, after checking that they satisfy the threshold.

Theorem 1 in Section IV shows that *Threshold1* has cost less than 2 times that of any algorithm that explores the entire part of the cross-product that can produce result tuples. *Threshold1* performs much better than this factor when the confidence distributions in the input relations are asymmetric.

An algorithm for *Threshold* with a better guarantee can be devised if the shape of the shaded region is known at the start. Suppose for example that we have meta-data (call it $D$) allowing us to find the confidence values of tuples at offsets that are multiples of $M$, as in Figure 4. We now describe an algorithm called *Threshold2* that uses this information while exploring the space.

Like *Threshold1*, *Threshold2* explores the pruned space, but this time in a more efficient manner. The idea is to choose the outer for the *explore* operation at each step depending on which relation allows for a "longer" rectangle. The choice is made by using meta-data $D$ to determine the approximate scan lengths for both possible outers. The exploration is detailed in Algorithm 2 and illustrated in Figure 6. The function *scan-lengths* returns the approximate scan lengths for both choices of outer using the meta-data $D$. For the example of Figure 4 with threshold 0.55, the order in which the blocks would be explored is $a, f, k, p, b, g, l, c$. As before, some blocks, namely $p, l, c$ are explored only partially. As in *Threshold1*, result tuples are emitted as they are found after checking that they satisfy the threshold.

**Algorithm 2** Threshold2

```
$o_{r_1} \leftarrow 0; o_{s_1} \leftarrow 0$
**while** 1 **do**
  **if** $conf(R, S, o_{r_1}, o_{s_1}) \leq \tau$ **then**
    $break$
  $l_r, l_s \leftarrow$ *scan-lengths*$(o_{r_1}, o_{s_1}, \tau, D)$
  **if** $l_r \leq l_s$ **then**
    *explore*$(R, S, o_{r_1}, o_{r_2}, o_{s_1}, \min(M + o_{s_1}, o_{s_2}))$
        $\{o_{s_2} \leftarrow \min_p conf(R, S, o_{r_1}, p) \leq \tau\}$
        $\{o_{r_2} \leftarrow \min_p conf(R, S, p, o_{s_1}) \leq \tau\}$
    $o_{s_1} \leftarrow o_{s_2}$
  **else**
    *explore*$(S, R, o_{s_1}, o_{s_2}, o_{r_1}, \min(M + o_{r_1}, o_{r_2}))$
        $\{o_{s_2} \leftarrow \min_p conf(R, S, o_{r_1}, p) \leq \tau\}$
        $\{o_{r_2} \leftarrow \min_p conf(R, S, p, o_{s_1}) \leq \tau\}$
    $o_{r_1} \leftarrow o_{r_2}$
```

Theorem 3 in Section IV shows that *Threshold2* has cost less than 1.5 times any algorithm that explores the entire part of the cross-product that can produce result tuples. This factor occurs only in cases where the part of the cross-product explored just exceeds the memory available. As the area to be explored becomes large compared to available memory, the factor comes closer to 1, as shown by Theorem 2 in Section IV.

## D. *Top-k*

A *Top-k* query returns $k$ result tuples with the highest confidence for a query-specified $k$. In this section, we discuss our algorithm for *Top-k* queries over two-relation joins. The idea again is to prune the space to be explored using a threshold as in the algorithms for *Threshold*. The threshold to be used is the confidence value of the $k^{th}$ tuple in the result, i.e., the minimum confidence value among the *Top-k* result tuples. Of course, this value is not known at the start of the algorithm.

During the algorithm we maintain a current top-$k$ result set in a priority queue $K$. When a result tuple $t$ is generated with confidence greater than the confidence of the lowest-confidence result tuple $t'$ in $K$, $t'$ is replaced by $t$. To avoid too much extra exploration, we restrict *scan* operations to a bounded length, say $L$.

Our *Top-k* algorithm is detailed in Algorithm 3. Effectively, the algorithm breaks up the cross-product into rectangles (*explore* operations) of breadth $M$ and length $L$. The highest-confidence tuple that a rectangle can produce is found using the combining function on the lower-left corner point, and is referred to as the *maxconf* for the rectangle. The algorithm explores rectangles in order of descending *maxconf*. In the example of Figure 4, the blocks would be explored in the order $a, b, f, g, k, l, c, p$, and so on depending on $k$. The algorithm maintains a priority queue $Q$ of rectangles yet to be visited.

The threshold used to stop loads and scans, and to terminate the algorithm, is the confidence of the current $k^{th}$ tuple in $K$.

**Algorithm 3** Top-$k$

```
$o_R[e] \leftarrow 0; o_S[e] \leftarrow 0$
$\mathit{confidence}[e] \leftarrow 1$
$Q \leftarrow e$
while $1$ do
  $i \leftarrow \mathit{ExtractMax}(Q)$
  if $\mathit{confidence}[i] \leq \mathit{Bottom}(K)$ then
    $break$
  $o_{r_1} \leftarrow o_R[i]$
  $o_{s_1} \leftarrow o_S[i]$
  $\mathit{explore}(R, S, o_{r_1}, \min(L + o_{r_1}, o_{r_2}), o_{s_1}, \min(M + o_{s_1}, o_{s_2}))$
                $\{o_{s_2} \leftarrow \min_p \mathit{conf}(R, S, o_{r_1}, p) \leq \mathit{Bottom}(K)\}$
                $\{o_{r_2} \leftarrow \min_p \mathit{conf}(R, S, p, o_{s_1}) \leq \mathit{Bottom}(K)\}$
  $o_R[r] \leftarrow o_{r_1} + L; o_S[r] \leftarrow o_{s_1}$
  $\mathit{confidence}[r] \leftarrow \mathit{conf}(R, S, o_R[r], o_S[r])$
  $Q \leftarrow r$
  $o_R[u] \leftarrow o_{r_1}; o_S[u] \leftarrow o_{s_1} + M$
  $\mathit{confidence}[u] \leftarrow \mathit{conf}(R, S, o_R[u], o_S[u])$
  $Q \leftarrow u$
```

Let $\tau$ be the confidence of the $k^{th}$ result. Theorem 5 in Section IV proves that *Top-k* with $L = M$ has cost less than 3 times any algorithm that explores the space that can produce tuples with confidence $\geq \tau$. The factor improves to 2 as the region explored becomes much larger than available memory. For such regions, larger $L$ may further increase efficiency. In fact, the best approach may be to vary $L$ for different *explore* steps, as discussed in Section III-F.1.

Technically, our guarantees for *Top-k* hold only in cases where sufficient memory is not available, which is the environment we are targeting. However, we can combine our algorithm with an efficient *Top-k* algorithm $A'$ that relies on sufficient memory being available, e.g., [14]. Such algorithms also maintain a queue corresponding to our queue $K$. If algorithm $A'$ runs out of memory during processing, it is possible to switch to ours, continuing processing on queue $K$.

## E. Sorted-Threshold

We now describe an algorithm for *Sorted-Threshold*, which returns result tuples with confidence over a threshold $\tau$, sorted by confidence. *Sorted*, which returns all results sorted by confidence, corresponds to the special case where $\tau < 0$. The algorithm explores the same pruned space as in the *Threshold* problem, but in an order resembling *Top-k*. The algorithm is detailed in Algorithm 4. Like *Top-k*, it uses two priority queues: $K$ for temporarily maintaining result tuples, and $Q$ for maintaining information about the rectangles yet to be explored. The threshold used to terminate loads and scans, and to terminate the algorithm itself, is the input $\tau$. The *explore* operation pushes result tuples that satisfy the threshold into the queue $K$. The algorithm maintains a value $\gamma$ that is an upper bound on the confidences of all unseen result tuples at any stage during the running of the algorithm. Result tuples in $K$ with confidence $\geq \gamma$ are emitted in sorted order as $\gamma$ decreases.

**Algorithm 4** Sorted-Threshold

```
$o_R[e] \leftarrow 0; o_S[e] \leftarrow 0$
$\mathit{confidence}[e] \leftarrow 1;$
$Q \leftarrow e$
while $1$ do
  $i \leftarrow \mathit{ExtractMax}(Q)$
  if $\mathit{confidence}[i] \leq \tau$ then
    $break$
  $o_{r_1} \leftarrow o_R[i]$
  $o_{s_1} \leftarrow o_S[i]$
  $\mathit{explore}(R, S, o_{r_1}, \min(L + o_{r_1}, o_{r_2}), o_{s_1}, \min(M + o_{s_1}, o_{s_2}))$
                $\{o_{s_2} \leftarrow \min_p \mathit{conf}(R, S, o_{r_1}, p) \leq \tau\}$
                $\{o_{r_2} \leftarrow \min_p \mathit{conf}(R, S, p, o_{s_1}) \leq \tau\}$
  $o_R[r] \leftarrow o_{r_1} + L; o_S[r] \leftarrow o_{s_1}$
  $\mathit{confidence}[r] \leftarrow \mathit{conf}(R, S, o_R[r], o_S[r])$
  $Q \leftarrow r$
  $o_R[u] \leftarrow o_{r_1}; o_S[u] \leftarrow o_{s_1} + M$
  $\mathit{confidence}[u] \leftarrow \mathit{conf}(R, S, o_R[u], o_S[u])$
  $Q \leftarrow u$
```

*Sorted-Threshold* emits each result tuple as soon the $\gamma$ threshold is passed, so it is a non-blocking algorithm. The algorithm explores the space in the same order as *Top-k*, but we can make stronger guarantees. Theorem 6 in Section IV shows that *Sorted-Threshold* with $L = M$ has cost less than 2 times any algorithm that explores the space that can produce tuples with confidence $> \tau$. Parameter $L$ is further discussed in Section III-F.1.

## F. Discussion

In this section we cover a few properties of and extensions to our algorithms not discussed in their initial presentation.

*1) Parameter L:* Recall that our *Top-k* algorithm uses input parameter $L$ to restrict the maximum length of *scan* operations. Our efficiency guarantees are proved for $L = M$. However, other choices of $L$ may yield more efficient computations for some inputs: If the space that needs to be explored to compute the top $k$ results turns out to be much larger than memory $M$, then it is beneficial to use a larger $L$. Various methods can be employed to detect and exploit this situation. For example, if statistics are available, we might estimate in advance the space to be explored and adjust $L$ accordingly. $L$ can also be adjusted adaptively during the algorithm based on what the algorithm has seen so far. Specifically, each invocation of the *explore* operation can dynamically choose a different $L$. Also note that when $L \neq M$ is used, the algorithm is no longer symmetric, so we can incorporate dynamic selection of the outer relation in a similar fashion to algorithm *Threshold2* (Section III-C).

*Sorted-Threshold* also may choose to adjust parameter $L$, but the considerations are somewhat different. In this case, $L$ encapsulates a memory-cost trade-off: A larger $L$, while being more cost-efficient, may require a larger buffer for result tuples. Here too we can set $L$ in advance based on known statistics, or modify it adaptively based on observations during the algorithm's execution. As before, we can also incorporate dynamic selection of the outer relation in each *explore* step.

*2) Using Indexes:* Our algorithms do not assume any indexes on the join attributes, and instead use pruning based on result confidences. Nested-loop index join can be modified to devise algorithms for all four query types which also do pruning based on result confidences:

- Sorted-by-confidence access is used on outer relation, and an index on the join attribute is used to access the inner relation.
- A buffer for result tuples is maintained for *Top-$k$*, *Sorted* and *Sorted-Threshold*, like the algorithms described above.
- The confidence of the last tuple from the outer can be used to determine an upper bound on the confidence of all unseen result tuples because of monotonicity.

These algorithms are modifications of the *Threshold Algorithm* [8]. They are easily used in the context of uncertain databases because they don't need to load the outer into memory, and hence can't run out of memory.

*3) Use in Queries:* Our algorithms have all been specified for two-way join operations. Suppose we have $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_m$, and we wish to obtain the result of this multi-way join using *Threshold*, *Top-$k$*, or *Sorted-Threshold*. Our approach is fairly simple: Since *Sorted* (i.e., *Sorted-Threshold* with $\tau < 0$) provides access to its results in a non-blocking fashion sorted by confidence, we build a tree of two-way joins. All lower nodes in the tree perform the *Sorted* operation on their two operands, while the root performs the desired operation (*Threshold*, *Top-$k$*, or *Sorted-Threshold*) on its operands. (Sometimes portions of intermediate results need to be materialized, if the algorithm needs to scan them multiple times.) Operators also have the option of using the indexed version of the algorithms as described in Section III-F.2, if the join condition permits it. All algorithms that use pruning based on result confidences, require the input sorted by confidence. This may be provided either by storing the relations sorted by confidence, or using a sort operator in the query plan which allows the use of other operators in a query plan. Sorted by confidence in a special "interesting" order because it can yield efficiency benefits for a significant fraction of queries. These new algorithms introduce new query optimization challenges that we leave for future work.

*4) Non-independence and self-joins:* Our algorithms rely on a monotonic combining function. Frequently in probabilistic databases (when data is mutually independent) this function is multiplication over the domain $[0, 1]$, and that is what we have used in our examples and experiments. There are cases when simple multiplication can not be used as the combining function because of non-independence in data (input and/or intermediate); self-join is one instance of this non-independence. Our techniques for pruning the join can still be used in the presence of non-independence:

- As was shown in [5], 'safe plans" can be found for a class of queries over uncertain databases. A safe plan ensures that all intermediate results are independent, allowing the query processor to compute confidences for intermediate result tuples using the simple multiplication combining function. The operators proposed in this paper can be used directly in all safe plans.
- Combining functions over non-independent tuples tend to be expensive, so a number of methods have been introduced to address this problem, e.g., lineage-based confidence computation in [6], and Monte Carlo techniques in [5], [18]. Both of these methods allow cheap interval-approximations to the results of the combining function, which are then refined as needed. Our algorithms can be extended to operate on interval-approximations instead of exact results, minimizing both data retrieval cost and (expensive) computation of exact values.

*5) Generalizing:* We cast our presentation in the context of uncertain and probabilistic databases, since that is the area in which we work, and it forms the context for our implementation. However, just as with other related algorithms (to be discussed in Section VI, e.g., [14]), our algorithms are quite generic. For example, they can also be applied directly to any setting with a monotonic 'scoring" function, such as middleware systems [7], multimedia databases [3], [17] and IR systems.

## IV. Efficiency Guarantees

We prove guarantees on the efficiency of our algorithms by comparing against other algorithms for the same problem. Each algorithm we consider takes as input $I$ the following:

- Relations $R$ and $S$ with sorted access by confidence
- Arbitrary, possibly opaque, join condition $\theta$
- Monotonic confidence value combining function $f$
- Threshold $\tau$ for *Threshold* and *Sorted-Threshold*
- Result size $k$ for *Top-$k$*

Every algorithm emits an unordered set of tuples as its result for *Threshold*, and an ordered set of result tuples for *Sorted-Threshold* and *Top-$k$*. An algorithm is *valid* if it returns the correct result for all possible inputs.

For efficiency comparisons we only consider algorithms in a class we refer to as $\mathcal{A}$. Algorithms in this class are restricted to use the same access on the data and memory as our algorithms, in order to capture the same environment as ours. Algorithms in class $\mathcal{A}$ adhere to the following:

- The relations are accessed only through their sorted-by-confidence interface.
- Memory available for storing tuples is limited to $M$. Small constant amounts of additional memory may be used for hash-table overhead, maintaining $k$ result tuples for *Top-$k$* algorithms, and other bookkeeping.

- No special processing is performed based on the join condition $\theta$ (e.g., using a disk-based sort-merge is $\theta$ specifies an equijoin).

Let $Cost(A, I)$ represent the cost of algorithm $A$ on input $I$. Recall that our cost metric, which can be applied to any algorithm in $\mathcal{A}$, captures the total amount of data retrieved through the sorted-by-confidence interface.

The following five theorems encompass our efficiency guarantees for *Threshold* and *Top-k*. (*Sorted* and *Sorted-Threshold* are discussed after these theorems.) Each theorem compares one of our algorithms against any valid algorithm in class $\mathcal{A}$ for the same problem. Note that *Threshold1*, *Threshold2* and *Top-k*, are valid algorithms in $\mathcal{A}$ for their respective problems.

Our results are quite strong in the following sense. Our bounds on efficiency are based on problem instances: We show for each instance a comparison of our algorithm against any other valid algorithm on the same instance. This comparison is stronger than comparing algorithms on all instances—in our comparison, a different "best" algorithm may compete with ours on different instances.

We provide intuitive proof sketches for each theorem. Full proofs can be found in the Appendix.

*Theorem 1:* Let $\mathcal{A}_t$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Threshold* problem, and let $\mathcal{I}_t$ represent the set of all inputs. The following holds:

$$\forall_{I\in\mathcal{I}_t} Cost(\textit{Threshold1}, I) < 2 \cdot \min_{A\in\mathcal{A}_t} Cost(A, I)$$

For an input $I$ to the *Threshold* problem, consider the part of the cross-product of the input relations at which the combining function $f$ evaluates to more than the threshold $\tau$, for example the shaded region in Figure 2. All valid algorithms for the problem must *cover* the shaded region, i.e., they must evaluate the join condition $\theta$ at all points in the region.

We show that *Threshold1* has cost less than 2 times the cost of any algorithm that covers the region. Intuitively, one lower bound corresponds roughly to the area (in units $M^2$) of the shaded region. However, another lower bound is the *semi-perimeter* (in units $M$) of the stair-like shaded region, which is equal to the sum of the lengths of the shaded $x$ and $y$ axes. The latter bound can sometimes be greater than the former (intuitively, when shaded regions are very "narrow"), so these bounds are combined in our proof. In any exploration performed by *Threshold1*, all but one of the rectangles have breadth 1. Any rectangle with cost $\geq 2$ covers area at least 1, and those with cost $< 2$ contribute at least 1 to the semi-perimeter. This intuition is formalized to prove Theorem 1.

A bad case for algorithm *Threshold1* occurs when all $b$ *explore* rectangles (say there are $b$ of them) have length 1 in units $M$. Out of the $b \geq 2$ rectangles, $b-1$ have cost 2, and the last rectangle has cost 1, giving a factor $\frac{2b-1}{b}$. Such an example can occur only for asymmetric confidence distributions.

*Theorem 2:* Let $\mathcal{A}_t$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Threshold* problem, and let $\mathcal{I}_t$ represent the set of all inputs. The following holds:

$$\forall_{I\in\mathcal{I}_t} Cost(\textit{Threshold2}, I) < \frac{3}{2} \cdot \min_{A\in\mathcal{A}_t} Cost(A, I)$$

In fact, the factor $\frac{3}{2}$ occurs only for very small shaded regions. For larger shaded regions, *Threshold2* achieves better factors as formalized next. Let $s_I$ be the number of $M \times M$ blocks that intersect the shaded region for input $I$.

*Theorem 3:* Let $\mathcal{A}_t$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Threshold* problem, and let $\mathcal{I}_t$ represent the set of all inputs. The following holds:

$$\forall_{I\in\mathcal{I}_t} Cost(\textit{Threshold2}, I) < \frac{s_I + p - c}{s_I - c} \cdot \min_{A\in\mathcal{A}_t} Cost(A, I)$$

where constants $p$ and $c$ are such that $p \geq c$ and $s_I \geq \frac{p\cdot(p+1)}{2}$.

The value $s_I$ is a measure of the size of the region to be explored. The factor decreases as $s_I$ increases, tending to 1 as $s_I$ tends to $\infty$. The intuition for the guarantee of *Threshold1* also applies to both guarantees for *Threshold2*. Additionally, because of the greedy choice of outer in *Threshold2*, the $i^{th}$-from-last rectangle explored by *Threshold2* always has length at least $i - 1$. This intuition is formalized to prove the Theorems 3 and 2.

Now let us consider the *Top-k* problem. An input $I$ satisfies the *distinctness* property if confidences of all result tuples in $R \times S$ are distinct. For *Top-k*, we restrict attention to inputs that satisfy this property. No valid algorithm can have any constant-factor guarantee corresponding to our guarantees in the general case (i.e., without assuming distinctness). Consider for example inputs where all confidences are 1, $k = 1$, and only one arbitrarily placed pair of tuples satisfy the join condition.

Recall that algorithm *Top-k* has a parameter $L$, discussed in Section III-F.1. We prove the following two theorems for $L = M$.

*Theorem 4:* Let $\mathcal{A}_k$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Top-k* problem, and let $\mathcal{I}_k$ represent the set of all inputs that satisfy the distinctness property. The following holds:

$$\forall_{I\in\mathcal{I}_k} Cost(\textit{Top-k}, I) < 3 \cdot \min_{A\in\mathcal{A}_k} Cost(A, I)$$

For an input $I$ to the *Top-k* problem, consider the part of the cross-product at which $f$ evaluates to at least the confidence $\alpha$ of the $k^{th}$ result tuple, i.e., our usual shaded region with $\tau = \alpha$. In a manner similar to *Threshold2*, the factor 3 occurs only for small shaded regions, and decreases for larger regions as formalized next. Let $s_I$ be the number of $M \times M$ blocks that intersect the shaded region.

*Theorem 5:* Let $\mathcal{A}_k$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Top-k* problem, and let $\mathcal{I}_k$ represent the set of all inputs that satisfy the distinctness property. The following holds:

$$\forall_{I\in\mathcal{I}_k} Cost(\textit{Top-k}, I) < \frac{2 \cdot s_I - 1}{s_I - c} \cdot \min_{A\in\mathcal{A}_k} Cost(A, I)$$

where constant $c$ is such that $s_I \geq \frac{c\cdot(c+1)}{2}$.

As above, value $s_I$ is a measure of the size of the region to be explored. The factor above decreases as $s_I$ increases, tending to 2 as $s_I$ tends to $\infty$. Recall that the area and semi-perimeter are lower bounds for any covering. Most $M \times M$

blocks explored by *Top-$k$* have cost 2 and cover area 1 (in units $M^2$), or contribute length 1 (in units $M$). This intuition is formalized to prove the factor 3 in Theorem 5. The fraction of blocks not satisfying the conditions above decreases with increasing $s_I$, and is formalized to prove Theorem 4

To formalize efficiency guarantees for *Sorted-Threshold*, we consider a class of algorithms $\mathcal{A}_\mathcal{M}$ that is a super-set of $\mathcal{A}$: As in $\mathcal{A}$, $M$ memory is available for storing tuples, and a small constant amount of additional memory is permitted. In $\mathcal{A}_\mathcal{M}$, algorithms may also use unrestricted memory to store result tuples only. *Sorted-Threshold* is a valid algorithm in $\mathcal{A}_\mathcal{M}$, and is compared against all valid algorithms in $\mathcal{A}_\mathcal{M}$.

*Theorem 6:* Let $\mathcal{A}_s$ represent the set of all valid algorithms in $\mathcal{A}_\mathcal{M}$ for the *Sorted-Threshold* problem, and let $\mathcal{I}_s$ represent the set of all inputs. The following holds:

$$\forall_{I \in \mathcal{I}_s} \mathit{Cost}(\mathit{Sorted\text{-}Threshold}, I) < 2 \cdot \min_{A \in \mathcal{A}_s} \mathit{Cost}(A, I)$$

The intuition behind the proof is similar to Theorems 1–4. Any valid algorithm in $\mathcal{A}_\mathcal{M}$ needs to cover the part of the cross-product that can contribute results with confidences over the threshold $\tau$. Each $M \times M$ square with cost 2 contributes area 1, and others contribute their cost to the semi-perimeter.

## V. Experiments

To evaluate the performance characteristics and trade-offs in our algorithms, we conducted several experiments on a large synthetic data-set with a variety of confidence distributions. The main objectives and results of our experiments are summarized as follows.

- Our algorithms have theoretical efficiency guarantees ranging from 1.5 to 3 times a lower bound in the general case, as shown in Section IV. We wanted to determine how close our algorithms are to the theoretical lower bound in practice. One interesting outcome is that in practice, *Threshold1* and *Threshold2* are similarly close to the lower bound, except when the input confidence distributions are asymmetric, *Threshold2* is much closer than *Threshold1*.
- Our efficiency guarantees hold for all distributions of confidence values on the input relations. We wanted to see how different confidence distributions affect the performance of our algorithms. In general, we do not see a dramatic change with different distributions, except as they affect the actual result.
- As discussed in Section III-F.1, our intuition suggests that algorithm performance can be affected by input parameter $L$. We wanted to test empirically how $L$ affects performance for algorithms *Top-$k$* and *Sorted-Threshold*. We find that the default choice of $L = M$ appears to be a good one.
- One feature of algorithm *Sorted-Threshold* is its non-blocking nature. We were interested in determining how the cost of the algorithm varies with the number of result tuples delivered, across different distributions. (This measure becomes very relevant when *Sorted-Threshold*, or *Sorted*, is used in a context in which not all tuples are necessarily requested.) We have the nice outcome that cost is linear in the number of results produced.

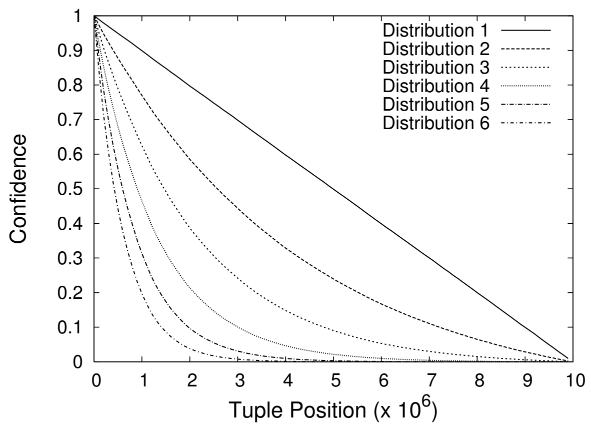

**Fig. 7:** *Confidence Distributions*

### A. Experimental Setup

We synthetically generate our two relations $R$ and $S$ with the following characteristics:

- Each data set has $|R| = |S| = N$ where $N$ is either one or 10 million tuples.
- The join between $R$ and $S$ is one-one in each data set. The joining pairs are independent of their relative confidence values, i.e., of their relative positions in sorted-by-confidence order.
- The confidence combining function is multiplication.
- Confidence distributions on each relation vary from uniform to highly skewed, while maintaining the positions of the joining pairs in sorted-by-confidence order. Figure 7 illustrates the six distributions we use in our experiments (denoted by $d \in \{1, 2, \cdots, 6\}$ in our graphs). The figure specifies the rate at which confidence values decrease in the sorted relation.

In the experiments, cost is measured as number of tuples retrieved from $R$ and $S$.

### B. Threshold

For the *Threshold* problem, we compare our two algorithms, *Threshold1* and *Threshold2*, against a lower-bound cost, as discussed in Section IV. We show four graphs. Figures 8 and 9 consider "symmetric" distributions in the two relations: we use Distribution 6 for both $R$ and $S$. Figures 10 and 11 consider "asymmetric" distributions: we use Distribution 1 for $R$ and Distribution 6 for $S$. We consider how performance varies with threshold $\tau$ in Figures 8 and 10, and with memory size $M$ in Figures 9 and 11. Fixed parameter values for experiments are noted at the top of each graph. Also, all costs are measured in terms of number of tuples retrieved.

In general both *Threshold1* and *Threshold2* are very close to the lower-bound (much closer than their theoretical guarantees). In particular, *Threshold2* nearly meets the lower-bound in all cases. *Threshold1* performs not quite as well when the relations have asymmetric distributions.

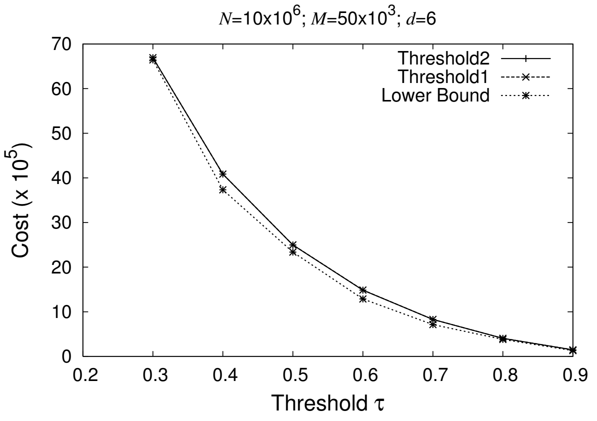

**Fig. 8:** *Cost of Threshold, symmetric distributions*

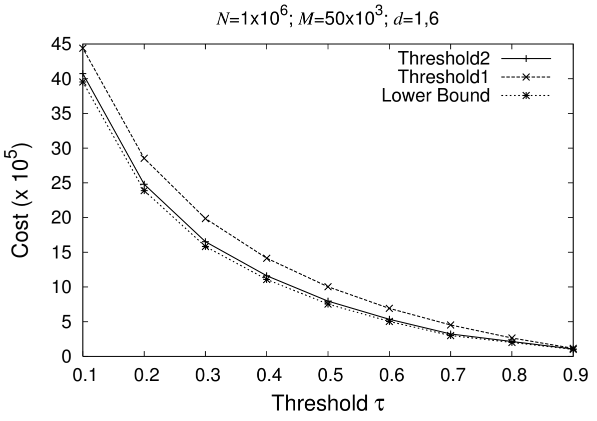

**Fig. 10:** *Cost of Threshold, asymmetric*

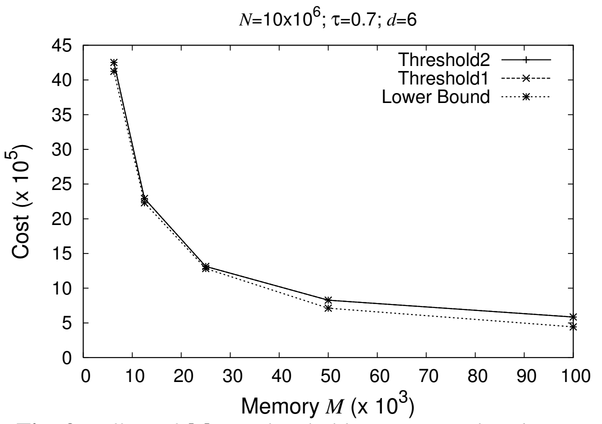

**Fig. 9:** *Effect of $M$ on Threshold, symmetric distributions*

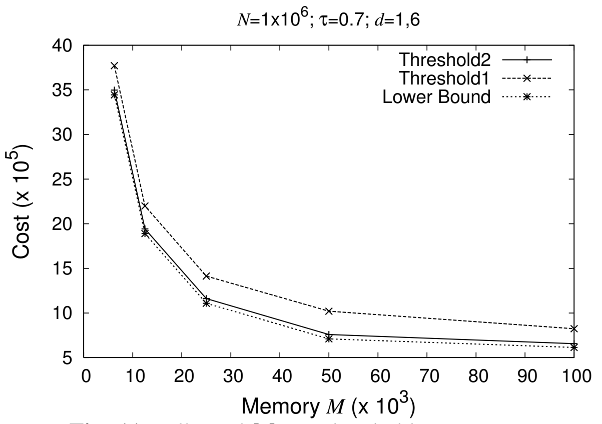

**Fig. 11:** *Effect of $M$ on Threshold, asymmetric*

### C. *Top-k*

For the *Top-$k$* problem we have four graphs:

- Figures 12 and 13 show how cost varies with $k$. (Figure 12 is for lower values of $k$, while Figure 13 "continues" Figure 12 with much higher values to confirm the trend.) We ran our *Top-$k$* algorithm to obtain its cost, noting the confidence of the $k$th result tuple. Using this confidence as threshold $\tau$, we plotted the cost of *Threshold2* and the lower-bound (as in the *Threshold* experiments; Section V-B). Note that the gap in performance between *Top-$k$* and the other two algorithms is expected since *Top-$k$* cannot know $\tau$ in advance. Also note that the cost of *Top-$k$* is significantly lower than the relation sizes.
- Figure 14 shows how the cost of *Top-$k$* is affected by memory size $M$. Again, we plot *Threshold2* and the lower-bound using the $\tau$ from the execution of *Top-$k$*.
- Figure 15 considers the effect of parameter $L$ on *Top-$k$* performance. Recall Section III-F.1, which discusses the setting of $L$: If we knew the confidence distributions and join selectivity in advance, we could select the "best" $L$. Figure 15 shows the cost when varying $L$ for a specific problem instance. As we see, $L = 100 \times 10^3$ is the best choice for this instance, although the default of $L = M = 25 \times 10^3$ also performs well.

Before looking at our results for *Sorted*, we consider the effect of confidence distributions on *Threshold* and *Top-$k$*. (We will separately consider the issue for *Sorted*.) Figure 16 plots the cost of *Threshold2* and *Top-$k$*, along with the lower-bound, for each of the six distributions shown in Figure 7 on both relations. (As in the *Top-$k$* graphs, *Threshold2* and the lower-bound use the $\tau$ from *Top-$k$*.) We see that the distribution has little effect on cost.

### D. *Sorted*

Our *Sorted* experiments all use the *Sorted-Threshold* algorithm with threshold $< 0$, i.e., they deliver a full sorted result. Our first conclusion, shown clearly in Figure 17, is that the cost of *Sorted* is proportionate to the number of results emitted. The two plots correspond to two different distributions, and behave nearly identically. Figures 18 and 19 consider different settings for parameter $L$, seeing how they affect cost and memory use, respectively, as tuples are emitted. Comparing the two graphs we see the trade-off clearly: a lower $L$ incurs higher cost but requires less memory. In Figure 19, the y-axis plots the largest buffer used so far in the execution. The horizontal steps correspond to the points at which a batch of result tuples is emitted.

## VI. Related Work

There has been a significant amount of previous work on top-$k$ queries in a ranking environment. For example, in the context of multimedia databases, a common goal is to apply a monotonic combining function to scores for different attributes

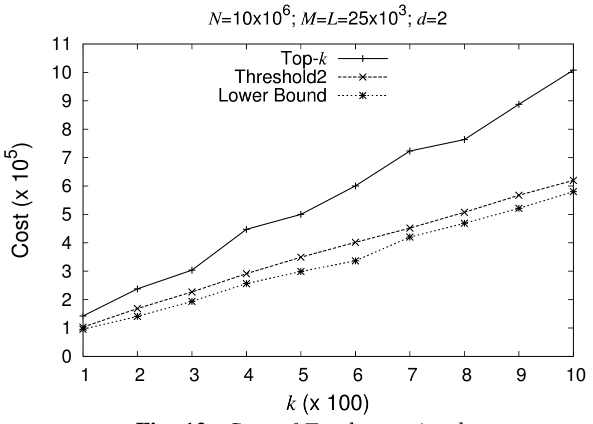

**Fig. 12:** *Cost of Top-k, varying k*

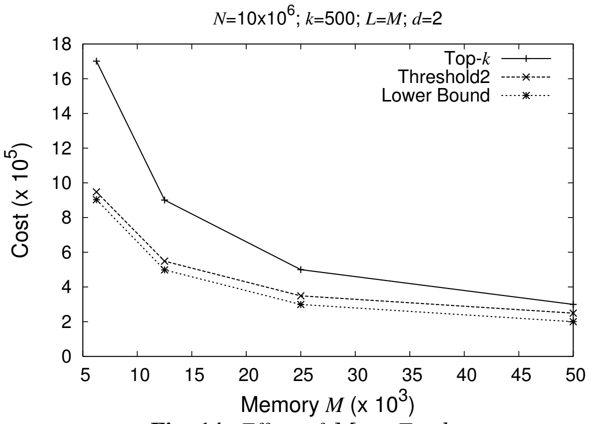

**Fig. 14:** *Effect of M on Top-k*

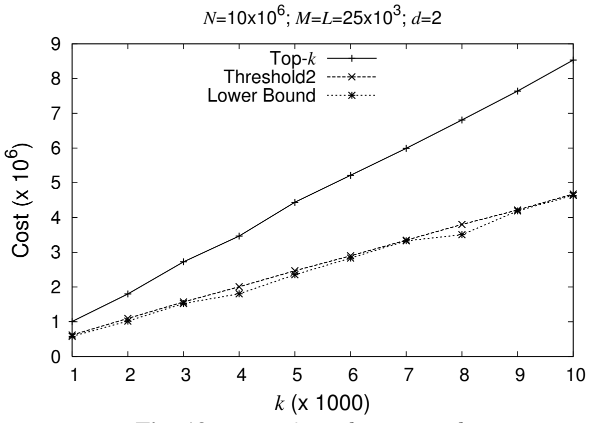

**Fig. 13:** *Cost of Top-k, varying k*

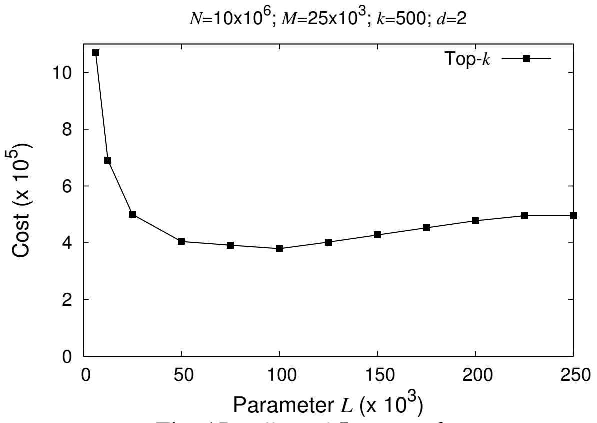

**Fig. 15:** *Effect of L on Top-k*

of an object, then select the top-$k$ objects by combined score. At a high level, this problem is isomorphic to the one we address, however there are several differences to our approach and goals, which we will highlight.

Many algorithms have been proposed for this problem. One distinguishing feature of previous algorithms is that many of them assume *random access* based on the identity of the objects being ranked. We assume only sorted access to our input relations. Previous algorithms that combine sorted and random access include *Fagin's Algorithm* [7], *TA* [8], *Quick-Combine* [11], and the "multi-step" algorithm presented in [17]. Algorithm *CA* in [8] specifically addresses the case where random access is expensive relative to sorted access, while [2] assumes only random access is available on some attributes. Similar ideas to the previous contributions are applied in [3], which shows how to use range queries to solve top-$k$ problems.

The fact that we consider an environment that does not permit random accesses distinguishes our algorithms from all of the references in the previous paragraph. A second significant distinguishing feature of our algorithms is that we operate in a memory-constrained environment. Although some previous algorithms, like ours, do not permit random accesses, all of them assume sufficient memory. Examples in this class include *NRA* [8] and *Stream-Combine* [12].

In a more conventional database setting, references [13], [14], [16] address the top-$k$ problem over join operations using only sorted access, very similar to our problem. Once again, unlike our work, the algorithms presented in these papers assume that sufficient memory is available. A problem tackled in [15] is how to optimize query plans containing "rank-aware" operators such as those in [13], [14]. Sufficient memory is still assumed in that work, however some of the query optimization ideas may be applicable in our setting.

There has been some recent work addressing top-$k$ or join queries in the context of uncertain databases [20], [18], [4]. The work in [18] does not address the problem of memory-efficient top-$k$ computations, but rather focuses on finding the top $k$ results efficiently when the computation of result confidence values (i.e., our combining function) may be very expensive. Their technique uses Monte Carlo simulations to refine intervals surrounding the result confidence values, with algorithms to minimize the amount of simulation required to find the top $k$ results. In Section III-F.5 we discussed how our algorithms might be useful in this setting. The work in [20] is very different from [18] and from ours: It defines and addresses new top-$k$ problems that arise due to the "possible worlds" semantics of uncertain databases. The work in [4] handles a problem that is very different from ours: It consider the problem of joins over uncertain attributes described as probability distribution functions.

## VII. Conclusions

The contribution of this paper is a suite of algorithms to address a new class of problems introduced by uncertainty

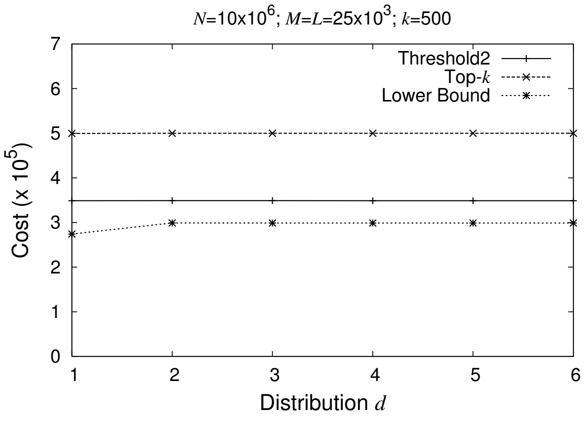

**Fig. 16:** *Effect of distribution on Threshold and Top-k*

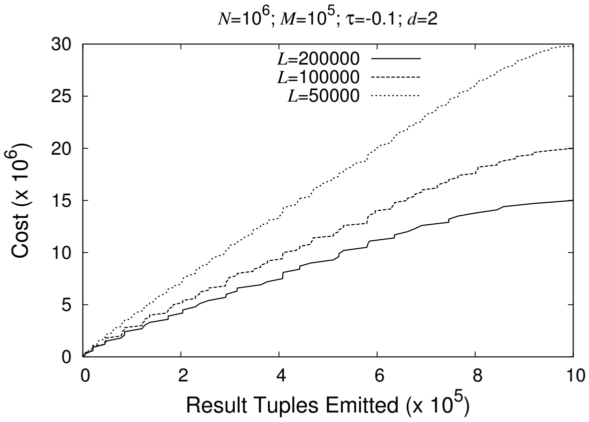

**Fig. 18:** *Effect of L on Cost of Sorted-Threshold*

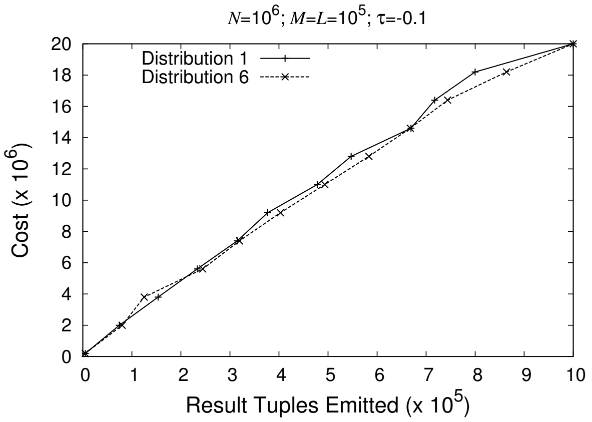

**Fig. 17:** *Emit Rate for Sorted-Threshold*

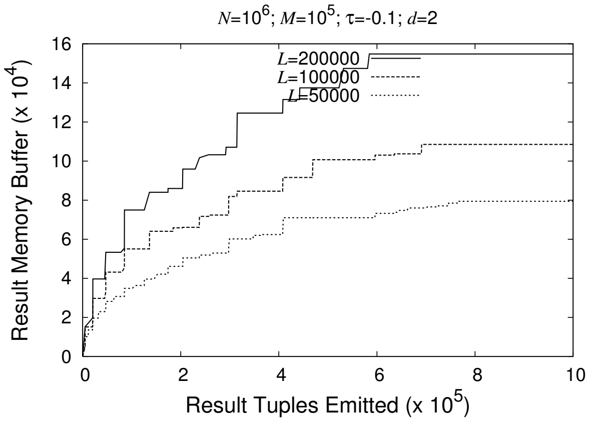

**Fig. 19:** *Effect of L on Memory Use of Sorted-Threshold*

in large databases. We consider queries that perform joins on uncertain relations with additional constraints based on result confidences: thresholding, top-$k$, or sorted. Unlike all previous algorithms for similar problems, we focus on an environment in which sufficient memory for processing during query execution may not be available. Similar to previous algorithms, we rely on access to input data sorted by confidence, and monotonicity of the function that computes results confidences.

For each of our four algorithms, *Threshold1*, *Threshold2*, *Top-k*, and *Sorted-Threshold* (with *Sorted* as a special case), we prove bounds on efficiency in comparison to any other algorithm for the same problem in the same memory-constrained environment. We have evaluated our algorithms empirically, showing closeness to lower-bound, sensitivity to problem parameters, and cost for non-blocking result delivery.

In terms of future work, as discussed in Section III-F.3, simple techniques can be used to extend our algorithms to handle multi-way joins, as opposed to the two-way joins we consider. However, we have not considered efficiency guarantees for this problem. Furthermore, a number of new issues arise for multi-way joins, such as memory allocation and join ordering. More generally, since our algorithms can be incorporated as new alternative operators in a general-purpose query processor, many new issues and opportunities arise in optimizing complex queries over large uncertain databases.

## References

[1] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[2] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[3] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *ACM SIGMOD*, 1996.

[4] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia. Efficient join processing over uncertain data. In *ACM CIKM*, 2006.

[5] N. Dalvi and D. Suciu. Efficient Query Evaluation on Probabilistic Databases. In *VLDB*, 2004.

[6] A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, 2008.

[7] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1), 1999.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4), 2003.

[9] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM TOIS*, 14(1), 1997.

[10] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. In *IIDB Workshop*, 2006.

[11] U. Guentzer, W.-T. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *VLDB*, 2000.

[12] U. Guentzer, W.-T. Balke, and W. Kiessling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, 2001.

[13] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, 2002.

[14] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.

[15] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *ACM SIGMOD*, 2004.

[16] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.

[17] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, 1999.
[18] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, 2007.
[19] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *ICDE*, 2007.
[20] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.
[21] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR*, 2005.

## Appendix

We first prove three Lemmas, which are then used to prove the six Theorems from the body of the paper. For an input $I$, consider a shaded region $S_I$ corresponding to points that can produce tuples with confidence greater than a threshold $\tau$. For *Top-k*, $\tau$ is the confidence of the $k^{th}$ result tuple.

*Lemma 1:* Any valid algorithm $A \in \mathcal{A}$ must evaluate the join condition at each point in $S_I$ for input $I$.

*Proof:* Suppose the join condition was not evaluated at point $p \in S_I$ by algorithm $A$. Consider input $I'$ that differs from $I$ only in the outcome of the join condition at point $p$. Algorithm $A$ cannot distinguish input $I$ from $I'$, and hence returns the same result. But $I$ and $I'$ have different correct results, hence algorithm $A$ cannot be valid. ■

Recall that $M$ represents the memory available to load tuples from the input relations. Let $s_I$ be the number of $M \times M$ blocks that intersect region $S_I$ corresponding to input $I$. For a block $b$, let $X_b$ represent the fraction of its lower edge that is within the shaded region. Correspondingly, let $Y_b$ represent the fraction of the left edge that is within the shaded region. Those blocks that have no block above or to the right intersecting the shaded region are referred to as *corner blocks* – let there be $c_I$ such blocks. We consider cost in units of $M$.

*Lemma 2:* Let algorithm $A \in \mathcal{A}$ evaluate the join condition at all points in region $S_I$. The following holds:

$$\mathit{Cost}(A, I) \geq s_I - c_I + \sum_{b \in corner} (X_b + Y_b)$$

where $s_I \geq \frac{c_I \cdot (c+1)}{2}$.

*Proof:* Any algorithm's execution can be viewed as a sequence of load and scan operations, which can be visualized like the *explore* steps in our descriptions. Note that the shapes formed in the visualization are not restricted to rectangles, but they are restricted to have width at most $M$. The cost of a shape is its semi-perimeter. We allocate the cost of shapes to the blocks they intersect in order to provide a lower bound for the cost of the algorithm. We consider the right and upper edges of a shape and allocate them to the block they lie in.

Formally, the restriction on the shapes is that for any interior point $p$, there is a right or upper edge within distance 1 of $p$. It is easy to observe that any block $b$ with $\max(X_b, Y_b) = 1$ has cost at least 1. Also for the corner blocks the cost is at least $X_b + Y_b$.

The relationship between $s_I$ and $c_I$ is obtained by observing that no row or column has more than 1 corner block. ■

Let $H_I$ and $W_I$ be the height and width respectively, of the two relations that the shaded region $S_I$ covers in units $M$.

*Lemma 3:* Let algorithm $A \in \mathcal{A}$ evaluate the join condition at all points in region $S_I$. The following holds:

$$\mathit{Cost}(A, I) \geq H_I + W_I$$

*Proof:* Any algorithm that evaluates the join condition at all points in $S_I$ must read the part of the each relation which is covered by $S_I$. The cost of doing this is $H_I + W_I$. ■

*Theorem 1:* Let $\mathcal{A}_t$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Threshold* problem, and let $\mathcal{I}_t$ represent the set of all inputs. The following holds:

$$\forall_{I \in \mathcal{I}_t} \mathit{Cost}(\mathit{Threshold1}, I) < 2 \cdot \min_{A \in \mathcal{A}_t} \mathit{Cost}(A, I)$$

*Proof:* Consider the cost of the *Threshold1* algorithm. The scan cost for all blocks except the rightmost in each row is 1. For every row except the topmost, the load cost is 1. Hence the following holds:

$$\mathit{Cost}(\mathit{Threshold1}, I) \leq s_I - 1 + Y_{lefttop} + \sum_{b \in rightmost} (X_b) \quad (1)$$

Using Lemma 1, and arithmetic manipulation using (1) and Lemmas 2 and 3 proves the theorem. ■

*Theorem 2:* Let $\mathcal{A}_t$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Threshold* problem, and let $\mathcal{I}_t$ represent the set of all inputs. The following holds:

$$\forall_{I \in \mathcal{I}_t} \mathit{Cost}(\mathit{Threshold2}, I) < \frac{s_I + p - c}{s_I - c} \cdot \min_{A \in \mathcal{A}_t} \mathit{Cost}(A, I)$$

where constants $p$ and $c$ are such that $p \geq c$ and $s_I \geq \frac{p \cdot (p+1)}{2}$.

*Proof:* Consider the cost of the *Threshold2* algorithm. Let it make $p_I$ greedy choices. The load cost is at most $p_I$. For each non-corner block the scan cost is at most 1, while for a corner block $b$, it is at most $\max(X_b, Y_b)$. The following holds:

$$\mathit{Cost}(\mathit{Threshold2}, I) \leq p_I + s_I - c_I + \sum_{b \in corner} \max(X_b, Y_b) \quad (2)$$

Clearly $p_I \geq c_I$, since a scan has to terminate at each corner block. Also the relationship between $s_I$ and $p_I$ is obtained by observing that the length of the scan corresponding to $i^{th}$-from-last scan is at least $i$. Using Lemmas 1 and 2 with (2) proves the theorem. ■

*Theorem 3:* Let $\mathcal{A}_t$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Threshold* problem, and let $\mathcal{I}_t$ represent the set of all inputs. The following holds:

$$\forall_{I \in \mathcal{I}_t} \mathit{Cost}(\mathit{Threshold2}, I) < \frac{3}{2} \cdot \min_{A \in \mathcal{A}_t} \mathit{Cost}(A, I)$$

*Proof:* For $s_I \geq 15$, Theorem 2 implies the result. For smaller $s_I$, case analysis proves the theorem using Lemmas 1, 2, and 3. ■

*Theorem 4:* Let $\mathcal{A}_k$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Top-k* problem, and let $\mathcal{I}_k$ represent the set of all inputs that satisfy the distinctness property. The following holds:

$$\forall_{I \in \mathcal{I}_k} \mathit{Cost}(\mathit{Top\text{-}k}, I) < \frac{2 \cdot s_I - 1}{s_I - c} \cdot \min_{A \in \mathcal{A}_k} \mathit{Cost}(A, I)$$

where constant $c$ is such that $s_I \geq \frac{c \cdot (c+1)}{2}$.

*Proof:* Consider the cost of the *Top-k* algorithm. It has cost at most 2 for each block except the last. For the last block $b$ it has cost at most $1 + \max(X_b, Y_b)$. The following holds:

$$\mathit{Cost}(\mathit{Top\text{-}k}, I) \leq 2 \cdot s_I - 1 + \max_{b \in corner} (\max(X_b, Y_b)) \quad (3)$$

Using Lemmas 1 and 2 with (3) proves the theorem. ■

*Theorem 5:* Let $\mathcal{A}_k$ represent the set of all valid algorithms in $\mathcal{A}$ for the *Top-k* problem, and let $\mathcal{I}_k$ represent the set of all inputs that satisfy the distinctness property. The following holds:

$$\forall_{I \in \mathcal{I}_k} \mathit{Cost}(\mathit{Top\text{-}k}, I) < 3 \cdot \min_{A \in \mathcal{A}_k} \mathit{Cost}(A, I)$$

*Proof:* For $s_I \geq 12$, Theorem 4 implies the result. For smaller $s_I$, case analysis proves the theorem using Lemmas 1, 2, and 3. ■

*Theorem 6:* Let $\mathcal{A}_s$ represent the set of all valid algorithms in $\mathcal{A}_{\mathcal{M}}$ for the *Sorted-Threshold* problem, and let $\mathcal{I}_s$ represent the set of all inputs. The following holds:

$$\forall_{I \in \mathcal{I}_s} \mathit{Cost}(\mathit{Sorted\text{-}Threshold}, I) < 2 \cdot \min_{A \in \mathcal{A}_s} \mathit{Cost}(A, I)$$

*Proof:* Consider the cost of the *Sorted-Threshold* algorithm. It has cost at most $X_b + Y_b$ for each block $b$. This gives us the following:

$$\mathit{Cost}(\mathit{Sorted\text{-}Threshold}, I) \leq 2 \cdot (s_I - c_I) + \sum_{b \in corner} (X_b, Y_b) \quad (4)$$

Using Lemmas 1 and 2 with (4) proves the theorem. ■