Supporting Incremental Join Queries on Ranked Inputs

Apostol Natsev Duke University natsev@cs.duke.edu

Yuan-Chi Chang IBM T. J. Watson yuanchi@us.ibm.com

John R. Smith IBM T. J. Watson jsmith@us.ibm.com

Chung-Sheng Li IBM T. J. Watson csli@us.ibm.com

Jeffrey Scott Vitter Duke University jsv@cs.duke.edu

Abstract

This paper investigates the problem of incremental joins of multiple ranked data sets when the join condition is a list of arbitrary user-defined predicates on the input tuples. This problem arises in many important applications dealing with ordered inputs and multiple ranked data sets, and requiring the top k solutions. We use multimedia applications as the motivating examples but the problem is equally applicable to traditional database applications involving optimal resource allocation, scheduling, decision making, ranking, etc.

We propose an algorithm J^* that enables querying of ordered data sets by imposing arbitrary *user-defined join predicates*. The basic version of the algorithm does not use any random access but a J^*_{PA} variation can exploit available indexes for efficient random access based on the join predicates. A special case includes the join scenario considered by Fagin [1] for joins based on identical keys, and in that case, our algorithms perform as efficiently as Fagin's. Our main contribution, however, is the generalization to join scenarios that were previously unsupported, including cases where random access in the algorithm is not possible due to lack of unique keys. In addition, J^* can support *multiple join levels*, or nested join hierarchies, which are the norm for modeling multimedia data. We also give ϵ-approximation versions of both of the above algorithms. Finally, we give strong optimality results for some of the proposed algorithms, and we study their performance empirically.

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

1 Introduction

Advances in computing power over the last few years have made a considerable amount of multimedia data available on the web and in domain-specific applications. The result has been an increasing emphasis on the requirements for searching large repositories of multimedia data. An important characteristic of multimedia queries is the fact that they return ordered data as their results, as opposed to returning an unordered set of answers, as in traditional databases. The results in multimedia queries are usually ranked by a *similarity score* reflecting how much each tuple satisfies the query. In addition, users are typically interested only in the top k answers, where k is very small compared to the total number of tuples. Work in this area has therefore focused primarily on supporting top-k queries ranked on similarity of domain-specific features, such as color, shape, etc.

One aspect that has received little attention, however, is the requirement to efficiently combine ranked results from multiple atomic queries into a single ranked stream. Most systems support some form of search based on a limited combination of atomic features but the combinations are usually fixed, limited to the types of features supported internally by the system, and typically allowing only Boolean combinations. In contrast, there has been little work on supporting top-k queries over arbitrary combinations, or joins, of multiple ordered data sets. One notable exception is Fagin's work [1, 2], who considered the case of equi-joins of ordered data when the join is over a unique record ID present in all of the join streams.

This paper formalizes the problem and generalizes it to user-defined join predicates. It also introduces a fast incremental J^* algorithm for that problem. The ability to support user-defined join predicates makes the algorithm suitable for join scenarios that were previously unsupported due to the lack of key attributes needed for random access. In the case when we do have such key attributes, or if we can exploit indexes to directly access tuples that satisfy user-defined predicates, we give a predicate access version of our algorithm that can take advantage of such indexes. We also consider approximation algorithms that trade output accuracy for reduced database access costs and space requirements. Finally, we study the algorithms empirically.

The rest of the paper is organized as follows. In the remainder of this section, we motivate and formally define the problem of top-k join queries on ordered data. We also review relevant work and list our specific contributions. We then give a detailed description of the J^* algorithm and its iterative deepening variation in Section 2. We explain the J^*_{PA} algorithm in Section 4, and discuss approximation algorithms in Section 5. In Section 6, we give some optimality results for the proposed algorithms. The behavior of the J^* algorithm is evaluated empirically in Section 7, where we validate our theoretical results in practice. We conclude with a summary of contributions.

1.1 Motivation

A good example to demonstrate the problem of joining multiple concepts with user-specified join constraints is the match of strata structures across bore holes in oil exploration services. Petroleum companies selectively drill bore holes in oil rich areas in order to estimate the size of oil reservoir underground. Readings from instruments measuring physical parameters, such as conductivity, as well as images of rock samples from the bore hole are indexed by depth, as in the example shown in Figure 1. Traditionally, experienced human interpreters (mostly geologists) will then label the rock samples and try to co-register the labels of one bore hole with those of many other bores holes in the neighboring area. The matched labels are then fed into 3D modeling software to reconstruct the strata structure underground, which tells the reservoir size.

The process of rock layer labeling and cross-bore coregistration can be significantly sped up by content-based image retrieval techniques. An interpreter can issue a query to locate similar looking rock layers in all bore hole images. An example is illustrated in Figure 1, where the query template consists of two rock types, and specifies that one should be near and on top of the other.

The above scenario is an example of combining the results of two ranked searches (*find occurrences of rock A and rock B*) based on user-defined constraints (*rock A is* **above** *rock B* and *rock A is* **near** *rock B*). We name such queries *ordered join queries with user-defined join predicates*, and we define them formally in the next section. Current database systems do not support such queries efficiently, and while multimedia search systems support some form of ranked join queries, to the best of our knowledge, none support join queries with user-defined join predicates.

Another feature that receives very limited support in both traditional and multimedia databases is the definition of nested (or recursive) views of ordered data sets. For example, building on the above oil exploration scenario, we can define the concept of $DELTA\_LOBE$ as a sequence of $SANDSTONE$ on top of $SHALE$ on top of $SILTSTONE$. The concepts of $SANDSTONE$, $SHALE$, and $SILTSTONE$, can be recursively defined by specifying samples of rock textures that fall in the corresponding class. Using the previously defined views for $SANDSTONE$, $SHALE$, and $SILTSTONE$, the user

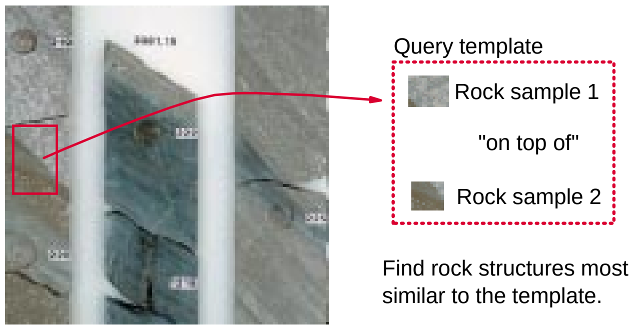


Figure 1: Example of an ordered join operation

might want to define the $DELTA\_LOBE$ view using the following SQL statement:

```
CREATE VIEW DELTA_LOBE AS
SELECT ∗
FROM    SANDSTONE SD, SHALE SH, SILTSTONE SL
WHERE above(SD.DEPTH, SH.DEPTH) = 1 AND
      above(SH.DEPTH, SL.DEPTH) = 1 AND
      near (SD.DEPTH, SH.DEPTH) = 1 AND
      near (SH.DEPTH, SL.DEPTH) = 1
```

Even though there is nothing conceptually new in this definition, in practice it is very hard to support such nested ordered views efficiently. The reason is that due to the imposed order, getting even a single candidate from the top view may involve materializing all candidates of the child views, which is very time consuming. Without an efficient join algorithm for ordered data, the database would be forced to perform a regular unordered join, followed by sorting, in order to get the single best answer in the parent view.

1.2 Definitions and Problem Formulation

In this section we consider the exact formulation of the join problem. The problem is illustrated in Figure 2. Informally, we are given m streams of objects ordered on a specific score attribute for each object. We are also given a set of p arbitrary predicates defined on object attributes from one or more streams. A valid join combination includes exactly one object from each stream subject to the set of join predicates (an example is denoted with a solid red line in Figure 2). Each combination is evaluated through a monotone score aggregation function defined on the score attributes of the individual objects, and we are interested in outputting the k join combinations that have the highest overall scores.

Our middleware cost model considers only the cost of accessing objects from the individual streams. We differentiate between two types of database accesses: *sorted access*, or scanning the objects in the order that they appear in each stream, and *predicate access*, or accessing all objects from a given stream that satisfy a certain predicate (e.g., return all objects that are within a certain distance of a fixed object). Note that *random access*, or accessing a specific object in a given stream, is a special case of predicate access where the predicate is the equivalence relation on an

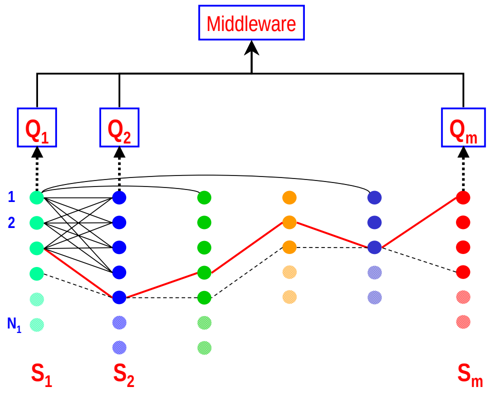


Figure 2: Cost model for the ordered join problem.

object's key attribute (e.g., its ID). Thus, our cost model is slightly more general than the one defined in [1, 2]. If C_S and C_{P_i} are the costs of scanning a single object using sorted access or predicate access (with predicate P_i), resp., and if N_S and N_{P_i} are the number of objects scanned in such manner, then the middleware cost is defined as:

$$Cost = N_S C_S + \sum_i N_{P_i} C_{P_i},$$

where i ranges over predicates used for accessing objects.

To formalize the problem using standard database terminology, we extend the definition of a *relational join*, or a *θ-join*, to the *ordered join* case as follows:[*]

Definition 1.1: *We define an* **ordered θ-join with respect to scoring function** *S as the combination of two tables ordered on a specific score attribute each and joined according to a specified relationship θ between one of more attributes in each table. The resulting table is ordered on an aggregate score attribute computed by the scoring function S:*

$$A \bowtie_\theta^S B = \{\, t = ab \mid a \in A, b \in B, \theta(a.X, b.Y) = 1, \\ t.score = S(a.score, b.score) \,\},$$

where A, B, and $A \bowtie_\theta^S B$ are ordered on their respective score attributes. ∎

We are now ready to formally define the problem:

Definition 1.2 [Ordered join top-k queries] *Given:*

- *Tables $A = \{a_i\}$ and $B = \{b_j\}$, ordered on a score attribute, and of size at most n records each;*
- *A score aggregation function $S : [0,1] \times [0,1] \longrightarrow [0,1]$ defined over the score attributes of the join tables that is monotone and incremental over each of its arguments;*
- *A set of Boolean predicates θ defined on one or more attributes from A and B.*

Output: Top k tuples from $A \bowtie_\theta^S B$ ∎

[*]This definition can be easily generalized to *m-ary joins*, or joins between m tables.

The *nested ordered join top-k query problem* is an instance of the above problem where at least one of Aor $B$is the result of another ordered join query. Otherwise, we will refer to the join as a *single-level join problem.* Note that the standard relational join corresponds to the special case where each of the tuples has a score of 1. Hybrid cases for joins of ordered and unordered data sets are therefore possible, with the unordered data being given an implicit score of 1.

When the set of join predicates θ contains only the equivalence relation on one or more attributes, the θ-join is called an *equi-join*. If in addition, the equi-join is defined on key attributes only, we call the resulting join *unique*. Unique ordered joins have been considered previously by Fagin et al. [1, 2], Ortega et al. [6], and Güntzer et al. [3]. However, the general class of joins based on arbitrary join predicates was previously unsupported efficiently, and is the main focus of this work.

1.3 Proposed Approach and Contributions

In this paper, we address the ordered join problem defined in Section 1.2. In addition to formulating the general problem precisely for the first time, we propose several algorithms for it, both exact and approximate. In contrast to a push model that requires blind scanning of the individual streams until a certain condition is satisfied, our algorithms use a pull model requesting inputs one at a time, and only if needed to determine the next best answer in the joined result set. This incremental computation is crucial for applying the algorithm to multi-level joins. To the best of our knowledge, the proposed algorithms are the first to efficiently support ordered joins based on arbitrary join predicates, as well as multi-level hierarchies of such joins. We present very strong optimality results for some of the algorithms, and we also perform an empirical study of their performance to validate their efficiency. Our specific contributions include algorithms for both database access scenarios defined in Section 1.2—using sorted access only or using both sorted and predicate access. In addition, we give approximation versions of the above-mentioned algorithms that provide guaranteed bounds on the approximation quality and can refine the solution progressively.

1.4 Related Work

The problem of supporting Boolean combinations on multiple ranked independent streams was first investigated by Fagin for the case of top-k queries [1]. The scenario he considered includes a database of N objects and m orderings (or ranked streams) of those objects according to m different ranking criteria (or atomic queries). The problem then consisted of evaluating the objects with respect to their ranks in each of the m streams, and outputting the k objects with the best overall scores. As such, the problem is equivalent to the unique equi-join special case of the general ordered join problem. In that case, the problem is that of combining scores that correspond to the same database object, as opposed to aggregating scores that belong to different objects.

Fagin proposed an algorithm for that problem that used both sorted access and random access based on unique key attributes [1]. Güntzer et. al. [3] optimized the original algorithm by formulating an earlier termination condition, and also considered optimizations for skewed input data sets. However, both algorithms relied heavily on random access, and in the more general join scenario, random access may be impossible due to lack of key attributes in the join constraints.

Very recently, we have become aware of new work by Fagin et al. [2], where the authors proposed three new algorithms for the unique equi-join scenario, including an algorithm that does not use random access. We note that our J^* algorithm is similar to that algorithm, although the two were derived independently, through different means, and have different interpretations. Also, they are still defined for different problem settings. In particular, we consider joining multiple sets of *different objects* under *arbitrary join constraints* that specify valid combinations of such objects. In contrast, the above algorithms all apply to the scenario of joining multiple sets of the *same objects* that are *ordered differently* in each stream.

Another treatment of the same problem of equi-joins over key attributes was presented by Ortega et. al in [6]. The authors defined a query tree whose nodes represented intermediate matches derived from the matches at the children nodes, and evaluated it bottom up to get the final similarity score at the root. They also proposed algorithm variations for specific score aggregation functions so that the algorithms would not use random access. Their methods, however, do not generalize to any monotone score aggregation function and to arbitrary join predicates.

The SPROC algorithm [4] is the only algorithm to the best of our knowledge that addresses the problem of joining ranked result sets under arbitrary join conditions. In the SPROC scenario, the overall score for the join combinations is a function of not only the *atomic scores* for each tuple but also *constraint scores* for pairs of tuples. The problem is therefore more general than the one considered here since it scores the extent to which the join predicates are met. It is solved by looking for a maximal cost path, computed with a Viterbi-like dynamic programming algorithm. In general, however, the algorithm will scan all streams completely due to the lack of monotonicity in the overall scoring mechanism. Therefore, it does not provide access cost savings but minimizes the number of evaluated join combinations instead.

One additional distinction of our work from all of the above works is the fact that the latter considered only the single-level join problem. In particular, random access makes the above algorithms inefficient for hierarchical joins since random accesses at intermediate levels of the join hierarchy are prohibitively expensive. Ortega et al.'s approach did use a multi-level query tree but the different levels were derived always by breaking up a single n-ary join into a hierarchy of binary joins. Neither of the previous approaches therefore considered nested views or ordered joins based on arbitrary join predicates.

2 Algorithm J^*

In this section we propose the J^* algorithm for the ordered join top-k query problem. The proposed join algorithm is based on the A^* class of search algorithms, hence the name J^*. The idea is to maintain a priority queue of partial and complete join combinations, ordered on upper bound estimates of the final combination scores, and to process the join combinations in order of their priorities. At each step, the algorithm tries to complete the combination at the top of the queue by selecting the next stream to join to the partial result and pulling the next tuple from that stream. The process terminates when the join combination at the head of the queue is complete. If that is the case, all incomplete join combinations will have scores smaller than the complete one at the head of the queue, and therefore, that combination corresponds to the next best answer. The algorithm thus performs the join incrementally, and due to its pull-based nature, it applies to the multi-level join hierarchies induced by nested view queries on ordered data.

More formally, for each input stream we define a variable whose set of possible values consists of the tuples from the corresponding stream. The problem of finding a valid join combination with maximum score reduces to the problem of finding an assignment for all the variables, subject to the join constraints, that maximizes the score. Therefore, define a *state* to be a set of variable assignments, and call the state a *complete* or *final solution* if it instantiates all variables. Otherwise, the state is called *partial* or *incomplete*. Since the possible values for each variable correspond to tuples with scores, we can define the score of a state assigning all variables to be simply the aggregation of the individual scores. For states that are complete, the score is exact. For incomplete states, the score can be upper bounded by exploiting the monotonicity of the score aggregation function. We then define the *state potential to be the maximum score a solution can take if it agrees with all assignments in the given state.* The state potential can be computed by upper bounding the scores of all non-instantiated variables for a given state. We can now solve the problem by running the A^* search algorithm, which mandates that states be processed in decreasing order of their potential.

The pseudo code for the crux of the algorithm is listed in Figure 3. In order to output the top k matches, the algorithm invokes the GetNextMatch() routine k times. During the main processing loop, the algorithm always processes the head of the queue by expanding it into two new states, generated by considering the next possible value for some unassigned variable. Both new states are inserted back into the priority queue according to their potential if they satisfy the join constraints. The GetNextUnassigned() routine encapsulates a heuristic that controls the order in which free variables are assigned. In general, we want to select the

```
GetNextMatch():
1   if ($queue$.Empty()) then
2       return NULL
3   endif
4   $head \leftarrow queue$.Pop()
5   if ($head$.Complete()) then
6       return $head$
7   endif
8   $head2 \leftarrow head$.Copy()
9   $head2$.AssignNextMatch()
10  if ($head2$.Valid()) then
11      $queue$.Push($head2$)
12  endif
13  $head$.ShiftNextMatch()
14  $queue$.Push($head$)
15  Goto Step 1
```

```
ShiftNextMatch():
1   $child \leftarrow$ GetNextUnassigned()
2   $child.match\_ptr$ ++
3   if ($child.match\_ptr ==$ NULL ) then
4       $child.match\_ptr \leftarrow$
5         $child$.GetNextMatch()
6   endif
```

```
AssignNextMatch():
1   $child \leftarrow$ GetNextUnassigned()
2   $child.match\_ptr$ ++
3   if ($child.match\_ptr ==$ NULL ) then
4       $child.match\_ptr \leftarrow$
5         $child$.GetNextMatch()
6   endif
```

Figure 3: Pseudo-code for the J^* join algorithm. The three functions above form the crux of the algorithm.

variable that will provide the largest refinement on the state potential, because the tighter the score upper bounds are, the faster the algorithm will converge to a solution. Examples of possible heuristics include selecting the variable that is most constrained or least constraining given the join predicates; the variable that *could* lead to the largest drop in the state potential; or the variable that *is expected* to lead to the largest drop.[†] In general, a heuristic that quickly reduces ambiguity in the partial solution should lead to faster convergence to the best solution, and the rate at which ambiguity is reduced by different heuristics can be measured empirically.

One important property to note from the pseudo-code is the recursive invocation of method GetNextMatch() from methods ShiftNextMatch(), and AssignNextMatch(). This illustrates why the algorithm works well on join hierarchies—when computing the next best match for a join at level l, the algorithm recursively invokes the same subroutine for some of the children at level $l+1$, where increasing levels correspond to deeper levels in the tree. This recursive mechanism enables nested joins/views to be processed efficiently. The efficiency comes from the fact that the recursive call is executed only if needed, using a pull-based model (i.e., a demand-driven approach).

The above algorithm is illustrated in Figure 4 for the oil exploration example introduced in Section 1.1. Suppose that we have two streams to join and the score aggregation function is simply the weighted average of the two scores, with weights 0.7 and 0.3. Stream A contains three possible matches, and stream B has only two matches. Suppose also that the "on-top-of" constraint is satisfied only for matching combinations $(A1, B1), (A2, B1), (A3, B2)$. The final solutions are listed in the top left corner of the figure, while the top right corner shows the first four iterations of priority queue processing (until the top answer is identified). Each state node in the priority queue contains a score estimate and the ID of the next possible match from stream A or B, resp. The unshaded squares denote unassigned variables, along with the corresponding score upper bounds, while the shaded squares contain variable assignments and exact scores.

Figure 4 illustrates the algorithm for a single join level only. If we had multiple join levels, the matches returned for the AB node, for example, would become part of the priority queue at an upper level. In that case, the steps in Figure 4 will be executed each time the parent node needs to get the next match for the AB node.

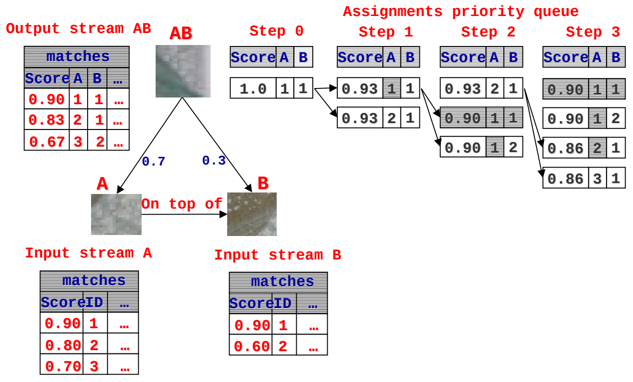


Figure 4: Illustration of the algorithm for query: *Find occurrences of rock texture A on top of rock texture B*. The individual matches for each query are shown in a table next to the corresponding view node. The matches for the root query are derived from the matches of the two children. The first few steps of the process are illustrated to the right by showing the priority queue at each of the steps.

[†]The last heuristic can be implemented by comparing the variable weights and their score distribution gradients. This observation was made in [3], where the authors achieved significant speedup for skewed score distributions using the same technique. They computed the weight by taking a derivative of the score aggregation function with respect to the given stream score, and approximated the gradient with a score difference. In some cases, however, the derivative may not be defined for the specific aggregation function and the weight may not be easily computable.

3 Algorithm J^* With Iterative Deepening

As defined in the previous section, the state potential can be expressed as a combination of the exact gain for reaching the given state and a heuristic estimate of the potential gain in reaching a terminal solution from the given state. By processing states in decreasing order of their potential, J^* automatically inherits all properties of an A^* algorithm.[‡] As long as the heuristic gain approximation never underestimates the true gain (i.e., the potential is always upper-bounded), A^* algorithms are guaranteed to find the optimal solution (i.e., the one with maximum gain) in the fewest number of steps (modulo the heuristic function). Therefore, they provide a natural starting point for any search problem [7].

A^* algorithms, however, are designed to minimize the number of processed states, not the database access cost from our cost model. Minimizing the access cost translates into minimizing the number of values considered for each variable assignment, and is not necessarily optimized by A^* algorithms. In addition, A^* algorithms suffer from large space requirements (exponential in the worst case), which also makes them less suitable in practice. In this section, we address both of the above issues by incorporating *iterative deepening* into J^*.

Iterative deepening is a mechanism for limiting computational resources by dividing computation into successive *rounds*. Each round has an associated cut-off threshold, called *depth* here, which identifies the round's boundaries. The depth threshold is defined on a certain parameter, and computation in each round continues as long as the value of that parameter is below the specified threshold. At the end of a round, if a solution has not been found, the algorithm commences computation in the next round. Iterative deepening can therefore be applied to a variety of algorithms by specifying the depth parameter and the threshold bounds for each round. Solution correctness and optimality are guaranteed as long as the modified algorithm can guarantee that solutions in earlier rounds are better in some sense than the ones in later rounds.

For our purposes, we can define the depth of an algorithm to be the maximum number of objects scanned via sorted access in any of the streams. We can define the depth of a state in a similar fashion, by counting the maximum number of objects considered for each variable in the given state. This definition for depth is very natural given that the cost of the J^* algorithm is directly proportional to the number of objects scanned via sorted access. We can further define the ith round to include all computation from depth $i \cdot s$ to depth $i \cdot s + s - 1$, inclusive, for some constant step factor $s \geq 1$. The step factor is needed to limit both access cost and space requirements in some worst cases, and can be used to control a tradeoff between the two. The pseudo-code for the modified J^* algorithm with iterative deepening is illustrated in Figure 5.

[‡] A^* algorithms form a class of search methods that prune the search space by taking into account the exact cost of reaching a certain state and a heuristic cost approximation of reaching a solution from the given state.

```
ITERATIVEDEEPENING-$J^*$():
1  Let $queue(r)$ be the priority queue at round $r$
2  $r \leftarrow 1$
3  $queue(1) \leftarrow$ root state
4  while $(queue(r).head \neq NULL$ and
5          $!queue(r).head.$COMPLETE$())$
6  do
7      if ($\exists$ free variable at depth $< (r+1)s$) then
8          process that variable
9      else
10         move $queue(r).head$ into $queue(r+1)$
11     endif
12 endwhile
13 if ($queue(r).head \neq$ NULL ) then
14     move $queue(r).head$ into $queue(r+1)$)
15 endif
16 $r \leftarrow r+1$
17 if ($queue(r).head ==$ NULL or
18     $queue(r).head.$COMPLETE$()$) then
19     return $queue(r).head$
20 else
21     Goto Step 4
22 endif
```

Figure 5: Pseudo-Code for the J^* algorithm with Iterative Deepening.

4 Algorithm J^*_{PA}

The algorithm from the previous section uses only sorted access when scanning the input streams. However, depending on the selectivity of the join predicates, it may be much more efficient to perform a predicate access if the system can exploit an index to return the objects that satisfy a given predicate. An extreme case is the random access scenario considered by Fagin [1], where each object participates in exactly one valid join combination (i.e., the probability of an arbitrary combination satisfying the join predicates is $\frac{1}{N^{m-1}}$). In that case, using random access to complete partial join combinations is much more efficient than scanning in sorted order until the join constraints are met.

We therefore propose a variation of the algorithm that can exploit indexes to directly access tuples based on the join predicates. The pseudo-code appears in Figure 6. The algorithm works like the J^* algorithm with one modification. When processing an incomplete state from the head of the priority queue, the algorithm first checks whether the state is instantiated sufficiently to allow completion by predicate access. If that is the case, the algorithm can process the state via predicate access rather than sorted access, provided the estimated cost of the predicate access is not too large. Note that each state processed via predicate access will be expanded into a number of new states (corresponding to all join combinations involving the returned objects from the uninstantiated streams). Therefore, we perform the predicate access only if the estimated number of returned objects is sufficiently small (i.e., smaller than a certain threshold). The threshold is determined dynami-

Algorithm J^*_{PA}:

```
1. Let $P$ be the set of join predicates
2. Call a state $\alpha$ *eligible* if
   $\forall$ non-instantiated stream $T$, $\exists$ a key predicate $p \in P$:
     (a) There is an index $I$ defined on $T$
     (b) $target(p)$ is a key column in $I$
     (c) $bound(p)$ is invariable given state $\alpha$
3. $sorted\_cost(\alpha) \equiv$ sorted cost when $\alpha$ reached
4. $predicate\_cost(\alpha) \equiv$ predicate cost when $\alpha$ reached
5. $credit(\alpha) \equiv sorted\_cost(\alpha) - predicate\_cost(\alpha)$
6. $cost(\alpha) \equiv \sum_{p \in P} C_p \cdot filter\_factor(p) \cdot N$
7. Run modified J*:
   If $head$ is eligible and $cost(head) \leq credit(t)$, then
     (a) Expand $head$ by predicate access
     (b) Insert resulting states into the priority queue
```

Figure 6: Pseudo-code for the J^*_{PA} join algorithm.

cally by the difference in sorted access cost vs. predicate access cost at that point in time.[§] The cost of a predicate access query (i.e., the number of returned objects) can be estimated with traditional selectivity estimation techniques (e.g., random sampling or histogram statistics). The decision on when to use predicate access can be based on traditional query optimization techniques.

5 Approximation Algorithms

Both algorithms J^* and J^*_{PA} solve the top-k query problem exactly. In some scenarios, however, the user may be willing to sacrifice algorithm correctness for improved performance. In the following, we describe an approximation version for both of the above algorithms. Intuitively, we call an algorithm an *ϵ-approximation* if it returns solutions that can be worse than optimal by at most ϵ. More formally, we adopt the definition from [2], and we say that an algorithm provides an ϵ-approximation to the top-k answers if it returns a set R of k solutions such that:

$$\forall x \in R, y \notin R: \quad (1+\epsilon) \cdot x.score \geq y.score.$$

Figure 7 illustrates the modified J^* and J^*_{PA} algorithms that return an ϵ-approximation for a user-specified ϵ. Alternatively, the algorithms can be modified to output the current approximation factor ($\epsilon = (U(k,t) - L(k,t))/L(k,t)$) at any given time t, and the user can decide when to stop the algorithm interactively. Note that for $\epsilon = 0$, both of the approximation versions behave exactly as the original algorithms and output the best k solutions.

[§] This heuristic essentially balances the two (negatively correlated) types of access cost in an effort to minimize the overall access cost. In [2], the authors present a hybrid algorithm, called CA, that balances sorted access cost with random access cost. Under certain assumptions, they prove optimality results independent of the unit costs for sorted access and random access. We believe that our algorithm has similar behavior.

Algorithm ϵ-J^* (resp., ϵ-J^*_{PA}):

```
1. Let $\epsilon \geq 0$ be a user-specified parameter.
2. Let $U(k,t)$ be the $k$th largest potential in the priority
   queue of $J^*$ (resp., $J^*_{PA}$) at time $t$ (i.e., $U(k,t)$ is an
   upper bound on the score of the $k$th best join combination).
   Let $U(k,t) = 1.0$ if the priority queue does not have $k$
   states at time $t$.
3. Let $L(k,t)$ be the $k$th largest potential of a complete state
   (i.e. solution) in the priority queue of $J^*$ (resp., $J^*_{PA}$)
   at time $t$ (i.e., $L(k,t)$ is a lower bound on the score of the
   $k$th best join combination). Let $L(k,t) = 0.0$ if the priority
   queue does not have $k$ complete states at time $t$.
4. Run iterative deepening $J^*$ (resp., $J^*_{PA}$) algorithm until
   time $t^*$ such that: $U(k,t^*) \leq (1+\epsilon)L(k,t^*)$.
5. Output the $k$ complete solutions with scores $\geq L(k,t^*)$.
```

Figure 7: Pseudo-code for the ϵ-approximation algorithms.

6 Optimality

In this section, we consider the performance of the proposed algorithms in terms of their database access cost, or cardinality of input tuples. We use the notion of *instance optimality*, defined by Fagin et al. in [2]:

Definition 6.1: *Let $\mathcal{A}$ be a set of algorithms that solve a certain problem, and let $\mathcal{D}$ be a set of valid inputs for that problem. Let $cost(A, D)$ denote the cost of running algorithm $A \in \mathcal{A}$ on input $D \in \mathcal{D}$. We say that algorithm $B \in \mathcal{A}$ is* **instance-optimal** *over $\mathcal{A}$ and $\mathcal{D}$ if $\forall A \in \mathcal{A}$ and $D \in \mathcal{D}$, $\exists$ constants c and c' such that:*

$$cost(B, D) \leq c \cdot cost(A, D) + c'.$$

The constant c is called the **optimality ratio** *of B.* ∎

As discussed in [2], the above notion of optimality is very strong if $\mathcal{A}$ and $\mathcal{D}$ are broad. Intuitively, if an algorithm is instance-optimal over the class of all algorithms and all inputs, that means that it is optimal in every instance, not just in the worst case or the average case. Given the above definition, we can state the following theorem:

Theorem 6.2 *Let $\mathcal{A}$ be the set of all algorithms that solve (resp., approximate) the top-k ordered join problem using only sorted access. Let $\mathcal{D}$ be the set of all valid inputs (i.e., database instances) for that problem. Then, algorithm J^* with iterative deepening (resp., ϵ-J^*) is instance-optimal over $\mathcal{A}$ and $\mathcal{D}$ with respect to database access cost. Furthermore, it has an optimality ratio of m, where m is the number of streams being joined, and no other algorithm has a lower optimality ratio.* ∎

Due to space consideration, we shall omit the proof of the above theorem. We only note that the lower bound on the optimality ration follows directly from [2], where Fagin et al. proved it for a special case of this problem.

7 Empirical Results

In this section we describe simulation experiments for evaluating the proposed algorithms. We implemented the algorithms as part of a constrained query framework that we proposed in [5]. The entire framework is about 5000 lines of C++ code and provides an API for plugging arbitrary attributes, constraints, and join algorithms. All of the experiments were run on an IBM ThinkPad T20, featuring a Pentium III 700 MHz processor, 128 MB RAM, and running Linux OS. All of the experiments we report use synthetic data sets, generated to model real data.

The queries were generated pseudo-randomly by having a fixed query tree structure but with random parameters, such as attribute values for the join predicates, node scores, and node weights. Each query was built as a tree joining m leaf nodes, where each leaf node had n matches. Each node performed an equi-join of its children views over a fixed attribute. The attribute values were generated randomly from an integer range that controls the probability that the join constraint is satisfied. Unless otherwise noted, we used probability of 0.5, in order to model general relational binary predicates (e.g., *left-of/right-of, above/below, smaller/bigger, before/after*, etc). Also, unless specified otherwise, children nodes had weights distributed uniformly in the [0,1] range and scaled to add up to 1. For the scores of matches at the leaf nodes (i.e., atomic queries), we considered the following distributions: uniform (i.e., linear decay), exponential decay, sub-linear decay, and the i%-uniform distributions from [3], for $i = 1, 0.1$, and 0.05. The i%-distributions were designed in [3] to model real image data and consist of i% of all scores being uniformly distributed in the [0.5, 1] range (i.e., medium and high scores), while the rest are all below 0.1 (i.e., insignificant scores).

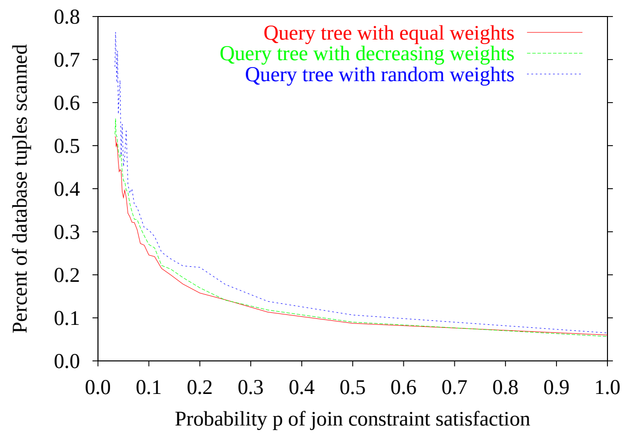
The parameters m, p, n, and the desired number of answers k for each query, are specified for each experiment. Default values are $m = 3$, $p = 0.5$, $n = 10000$, $k = 30$, and 1%-uniform score distributions. We performed experiments to study the performance with respect to constraint probability, query tree size, number of outputs, database size, stream weight and score distributions. To evaluate the performance of a query, we measured the number of tuples scanned by the algorithm, as well as the maximum size of the priority queue. The first measure is the database access cost of the algorithm, while the second corresponds to the space requirements of the algorithm. All of the results we report are averaged values over 10 random queries of the given type. The results are listed in Figures 8(a)–11.

The first set of experiments studies the dependence of the algorithm on the probability that the join constraints are met. The join constraints' probabilities were modeled by assigning a random attribute with d possible values to each node, and using pairwise join constraints that require the attributes to have the same value. We varied d from 1 to 30, thus obtaining probabilities $p = 1/d$ from 0.03 to 1.0. The results are plotted in Figures 8(a) and 8(b) for three different weight distribution cases—uniform, decaying, or equal weights. Figure 8(a) shows almost identical database access cost for all three cases, which means that the running time of the algorithm is fairly robust with respect to the weight distribution. Figure 8(b), however, shows that in the case of equal weights, the algorithm has a higher space requirement. This is to be expected since in that case the algorithm cannot exploit the weights to scan "more important" streams first, and will therefore take longer to converge to the optimal solution. Overall, both figures show that for reasonable values of p, when there are enough valid combinations to generate the desired number of outputs, both the database access cost and the space requirements are almost constant.

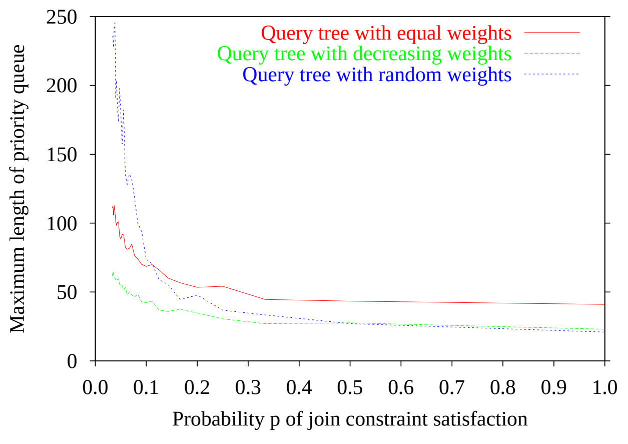
The second set of experiments evaluated the performance of the algorithm with respect to the size of the query tree. Given the number of streams to join, we considered two types of queries. The first was *flat* queries joining all input streams in a single level (denoted as *max-width* queries). The second were nested *max-height* queries that join the same number of streams but only two at a time, by building a balanced binary tree on the input streams. Figures 9(a)–9(b) show the performance of both types of queries with identical other parameters, and with varying number of streams to join. Note also that we used larger streams (100000 tuples) in these queries in order to test scalability with respect to database size. We can make several conclusions from the figures. First, despite the increased database size, both types of queries scan only a small number of tuples that appears to be dependent on the desired number of outputs only and not on the database size. And second, all else being equal, nested queries are more costly than flat queries in terms of access cost but cheaper in terms of space requirements. The higher access cost can be explained by the fact that the number of possible matches increases exponentially at each level in the nested queries. Yet, the difference in the access cost is fairly small, which shows that the algorithm is efficient even for such highly-nested joins.

The third set of experiments was designed to evaluate the dependence of the algorithm on different score distributions. We considered the six distributions described earlier and computed access cost and space requirements for varying number of desired outputs, k. The results in Figures 10(a) and 10(b) generally show a sub-linear dependence on k.[¶] An exception is the exponentially decaying score distribution, where the difference between successive score values becomes negligible very quickly and the algorithm takes longer to converge due to fact that successive assignments lead to very small refinements in the overall solution score. However, this trend is reversed for space requirements in Figure 10(b), where quickly decaying distributions require less space. This could also be explained by the theory that quickly decaying scores lead to small refinements in the score estimates very quickly, and therefore

[¶]Note that the number of scanned database tuples can be smaller than the number of desired outputs. This is due to the fact that each tuple can participate in multiple valid join combinations.

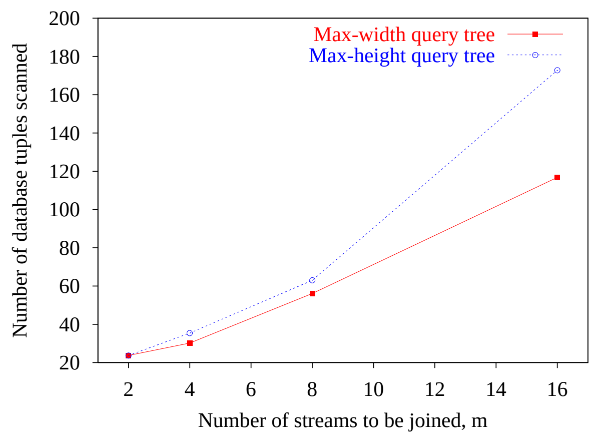


(a) Database access cost

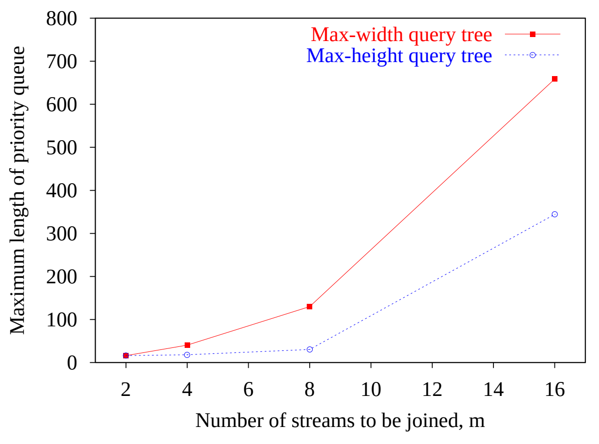


(b) Space requirements

Figure 8: J^*'s dependence on join constraint probability p. (1%-uniform distribution, $m = 3, n = 10000, k = 30$)

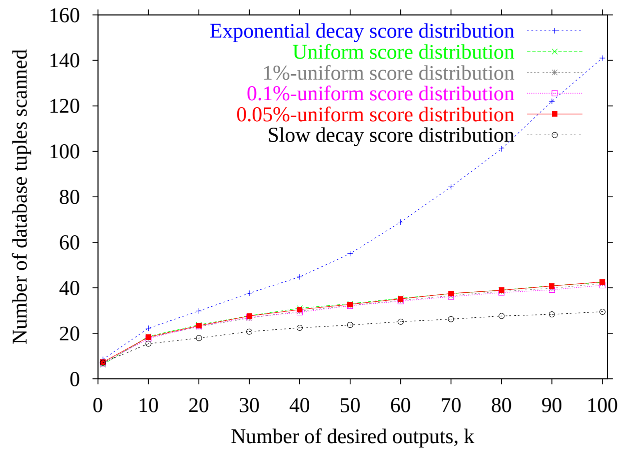


(a) Database access cost

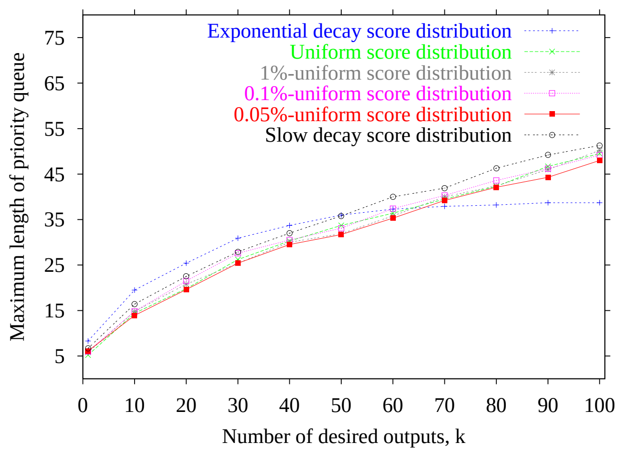


(b) Space requirements

Figure 9: J^*'s dependence on number of streams to join, m. (1%-uniform distribution, $n = 100000, k = 30, p = 0.5$)



(a) Database access cost



(b) Space requirements

Figure 10: J^*'s dependence on number of desired outputs, k. ($m = 3, n = 100000, p = 0.5$)

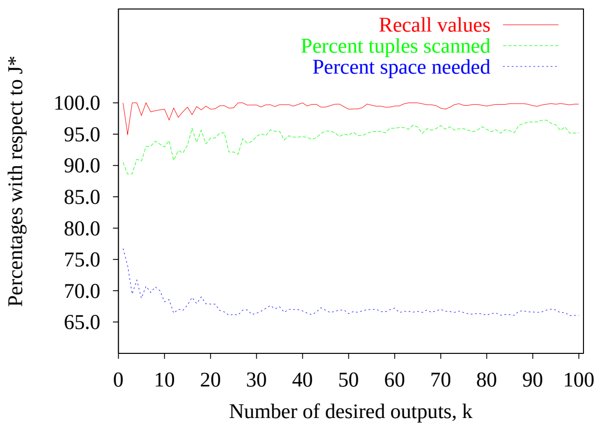


Figure 11: J^*-relative performance (in percentages) of *first-k* greedy J^* algorithm vs. number of desired outputs. (1%-uniform distribution, $m = 3, n = 100000, p = 0.5$)

the algorithm is more likely to be localized to the same set of assignments, as opposed to spreading its computation over a large set of assignments.

The final experiment measured the performance of a very simple *first-k* greedy approximation algorithm with respect to the original J^* algorithm. The idea is to run J^* until there are k complete and valid join combinations, and then to output them as the best solutions, even if they are not at the top of the priority queue. This corresponds to running the ϵ-J^* algorithm and stopping at the earliest possible time (i.e, for any ϵ approximation factor that is not ∞) in order to produce the coarsest approximation with that algorithm. We measured the database access cost and total space requirements of the greedy J^* version as a fraction of the corresponding values for J^*. We also calculated the recall and precision values. We considered the output of the J^* algorithm for a top-k query to be the ground truth for that query, and therefore, each top-k query had exactly k correct answers. Thus, the precision, defined as the fraction of output answers that were correct, is the same as the recall, or the fraction of correct answers retrieved by the approximation algorithm.

The recall, relative access cost and relative space cost are shown as percentages in Figure 11. From the recall curve in the graph, we can conclude that the greedy heuristic is an excellent approximation to the optimal answers for values of k that are not very small. We hypothesize that the greedy algorithm outputs the tuples in a slightly different order, which reduces the recall at the beginning. However, the identities of the top-k tuples eventually match the true answers, even though they might be shuffled somewhat. Thus, the *unordered* set of top-k answers is approximated very well. In addition, we see that the database access cost is reduced by 5–10%, while the space requirements are reduced by 40%. Therefore, we can conclude that the greedy *first-k* heuristic provides significant cost savings with almost no reduction of accuracy.

8 Conclusions

In this paper, we introduced several algorithms for incremental joins of ranked inputs based on user-defined join predicates. The algorithms enable more powerful querying by providing the ability to integrate result sets from multiple atomic independent queries, using complex criteria for integration. The need for such efficient query integration arises naturally in domains dealing with ordered inputs and multiple ranked data sets, and requiring the top k solutions.

Our proposed J^* algorithm differs from previous work in two main aspects: 1) it can support joins of ranked inputs based on *user-defined join predicates*; and 2) it can handle multiple levels of joins that arise in *nested views*. This is the first algorithm to the best of our knowledge that supports the above operations. We also presented a J^*_{PA} version of the algorithm that uses predicate access to reduce the cost of the algorithm, and we discussed variations for both scenarios that reduce complexity by approximating the solution. We gave strong optimality results for the algorithms requiring only sorted access. We also performed an extensive empirical study for validation in practice.

References

[1] R. Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58:83–99, 1999. An extended abstract of this paper appears in {*Proc. Fifteenth ACM Symp. on Principles of Database Systems (PODS '96)*}, *Montreal, 1996, pp. 216–226.*

[2] *R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In* Proc. of ACM Symposium on Principles of Database Systems (PODS '01), *Santa Barbara, CA, May 2001.*

[3] *U. Güntzer, W.-T. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In* Proc. of the 26th Intl. Conference on Very Large Databases (VLDB '00), *Cairo, Egypt, 2000.*

[4] *C. S. Li, J. R. Smith, V. Castelli, and L. Bergman. Sequential processing for content-based retrieval of composite objects. In* Storage and Retrieval of Image and Video Databases, VI. *SPIE, 1998.*

[5] *A. Natsev, J. R. Smith, Y.-C. Chang, C.-S. Li, and J. S. Vitter. Constrained querying of multimedia databases: Issues and approaches. In* Proc. SPIE Electronic Imaging 2001: Storage and Retrieval for Media Databases, *San Jose, CA, Jan. 2001.*

[6] *M. Ortega, Y. Rui, K. Chakrabarti, K. Porkaew, S. Mehrotra, and T. S. Huang. Supporting ranked Boolean similarity queries in MARS.* IEEE Trans. on Knowledge and Data Engineering, *10, Nov.–Dec. 1998.*

[7] *S. Russell and P. Norvig.* Artificial Intelligence: A Modern Approach. *Prentice Hall, Inc., 1995.*