Processing Top-k Join Queries

Minji Wu
Rutgers University
minji-wu@cs.rutgers.edu

Laure Berti-Équille[*]
University of Rennes 1
berti@irisa.fr

Amélie Marian
Rutgers University
amelie@cs.rutgers.edu

Cecilia M. Procopiuc
AT&T Labs-Research
magda@research.att.com

Divesh Srivastava
AT&T Labs-Research
divesh@research.att.com

ABSTRACT

We consider the problem of efficiently finding the top-k answers for join queries over web-accessible databases. Classical algorithms for finding top-k answers use branch-and-bound techniques to avoid computing scores of all candidates in identifying the top-k answers. To be able to apply such techniques, it is critical to efficiently compute (lower and upper) bounds and expected scores of candidate answers in an incremental fashion during the evaluation. In this paper, we describe novel techniques for these problems.

The first contribution of this paper is a method to efficiently compute bounds for the score of a query result when tuples in tables from the "FROM" clause are discovered incrementally, through either sorted or random access. Our second contribution is an algorithm that, given a set of partially evaluated candidate answers, determines a good order in which to access the tables to minimize wasted efforts in the computation of top-k answers. We evaluate our algorithms on a variety of queries and data sets and demonstrate the significant benefits they provide.

1. INTRODUCTION

While search engines are becoming increasingly good at returning the most relevant pages for a set of keywords, they are less able to integrate information from multiple sources in a well-structured way. For wide-interest domains - the so-called "verticals" - a certain degree of integration is built into the engines. Information relevant to that field is downloaded from multiple sources and joined inside the search engine's index. However, the process of deciding which attributes to extract and integrate is mostly manual, and the approach does not extend to more obscure areas of human interest.

We focus on efficiently computing answers for join queries that involve Web-accessible databases. Consider the join graph shown in Figure 1 as an example. Assume that edges represent tables of user interest extracted from the web and nodes represent attributes on which two tables join. We consider that each tuple (*e.g.*, (a_1, b_1) from T_1) carries a score that represents the quality of the tuple. Intuitively, the score for a join of multiple tuples is computed as the product of the scores of each tuple. Given a value for an attribute (*e.g.*, a), a possible join query would be to retrieve the value of another attribute (*e.g.*, c) connected to the first attribute via a join path (*i.e.*, $e_1 \rightarrow e_2$). An important observation is that given a source and a destination attribute, there may exist multiple join paths connecting them. We consider that each join path *vouches* for the quality of the generated join result. Therefore, in addition to the attribute value in the destination node, the user may be also interested in the tuples (*i.e.*, edges) that support the join result. In this paper, we consider the answer (which we call a *binding*) to the join query as a combination of tuples from each participating table and compute the score for the binding based on the score of each tuple. We show a motivating example in Section 2 and formally define the *binding* in Section 3.2.

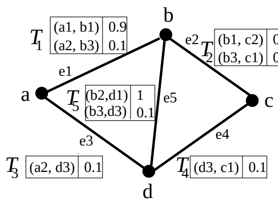


Figure 1: An example of join graph depicting the join relations between tables

In this paper, we consider the major bottleneck of top-k join query processing to be tuple accessing of web-accessible databases. If, for instance, the tables involved in one query are stored on different servers, and can only be accessed via a Web interface, executing a single join between two tables may become very expensive, as Web accesses exhibit high and variable latency. In addition, the query optimizer in one database will generally have no statistics about tables stored at remote sites and thus be unable to offer any improvements over the naive approach.

Top-k join query processing over ranked inputs has been studied in literature (*e.g.*, [3, 8]). Ilyas et al. [3] propose a rank-join algorithm that makes use of the individual orders of its inputs to produce join results ordered on a user-specified scoring function. Despite the performance advantage, the rank-join algorithm suffers from two limitations. First, the join queries considered in [3] involve tables that form only one join path. In contrast, we consider a more general join graph, allowing multiple join paths between the source and destination attributes.

The second limitation of rank-join algorithm is that it is essentially considering inner-join, which requires the join answer to have an instantiated tuple on each join edge along the join path. The study in [5] shows that inner join may produce answers with scores that are too low to be of interest. Consider the join graph in Fig-

[*]Mobility research program supported by the European Commission (Grant FP6-MOIF-CT-2006-041000)

ure 1. Assume we find one complete join answer with score 0.1 on each edge and another partial join answer with score 0.9 on edge e_1 and e_2 and `null` on all other edges. Clearly the latter join answer has a higher score and therefore is of more interest to the user. Even if the rank-join algorithm could be applied to the join graph considered in our case, it could not produce the latter answer since it contains `null` tuples on some of the join edges. For comparison purposes, we extended the rank-join approach to more general join graphs. Our experimental results show that our approach is significantly better than the rank-join based approach.

Our contributions. In this paper, we propose a novel branch-and-bound algorithm for computing the top-k answers for join queries over Web-accessible databases. Rather than computing all the results of the join query, our strategy dynamically retrieves a subset of tuples from each table, and maintains lower and upper scores bounds for the query results that include the retrieved tuples. By ordering the retrieval of table tuples based on the score bounds of the partial results, our algorithm results in significant savings in the number of Web accesses. We make the following contributions:

- We propose a model for scoring answers of arbitrary join graphs based on network reliability. We also develop methods for computing score bounds for partial answers.
- We present a novel branch-and-bound algorithm which aims to minimize the number of Web accesses required for computing the top-k answers.
- We evaluate our algorithms on a variety of queries and data sets and demonstrate the significant benefits they provide.

The rest of this paper is structured as follows. Section 2 presents a real-life example that we use in our experimental study. The example illustrates the concepts that we formally define in Section 3. Section 4 presents our dynamic probing techniques that efficiently compute the top-k results. We present our experimental study in Section 5. A brief review of related work appears in Section 6, and we conclude in Section 7.

2. ILLUSTRATIVE EXAMPLE

Suppose that a sophisticated marketer wants to design personalized promotional packages for attendees of certain scientific conferences. To optimize his strategy, he would like to find out who are the researchers most likely to attend which conferences, and what are their main reasons. The marketer decides that he could estimate the answer with reasonable accuracy by taking into account the following factors:

F1: Travel cost for each potential attendee to each conference site;

F2: Whether a potential attendee has at least one accepted paper; has a tutorial; is a conference organizer; or is a conference committee member.

F3: How important the conference is in its field.

F4: Whether the attendee is likely to attend in order to meet with a close collaborator such as his Ph.D. advisor; and how likely the collaborator is to attend.

The marketer finds several sites that each contains part of the data he needs. For example, a list of researchers' contributions to various conferences can be obtained from DBLife[1]. The same site also has information on researchers' affiliation, and thus their location. Travel sites return travel costs between any two locations. Conference locations can be obtained from the DBLP website, and IA Genealogy has a fairly large list of researchers' Ph.D. advisors.

[1]http://dblife.cs.wisc.edu/

Suppose that the following structured data is accessible from these websites.

- Table $Research$ with attributes $\{person, conf, \sigma\}$, where σ is the tuple score, normalized between 0 and 1: Tuples connect researchers to conferences. The value σ is a measure of the strength of this connection, based on their roles in that conference (author, tutorial giver, organizer etc.). For example, $(A, VLDB09, 0.9) \in Research$ may mean that researcher A will give a tutorial at VLDB09. Intuitively, this means he is very likely to attend VLDB09, so the tuple has a high score. Tuple $(A, ICDE09, 0.5)$ may mean that researcher A has one accepted paper at ICDE09, with another co-author.

- Table $Travel$ with attributes $\{person, loc, \sigma\}$: Tuples in this table reflect how cost-effective it is for a researcher to travel to a location. For example, $(A, Shanghai, 0.1)$ means that researcher A has only expensive options for traveling to Shanghai, while $(A, Providence, 0.9)$ means that researcher A has at least one cheap option for going to Providence; e.g., researcher A may live in New Jersey and travel by train.

- Table $People$ with attributes $\{person, advisor, \sigma\}$: Tuples in this table reflect the strength of the professional connection between a person and their advisor. This strength may be measured as, e.g., the percentage of papers a person co-authored with their advisor in the past 5 years; or as the inverse of the number of years since the person graduated.

- Table $Conference$ with attributes $\{conf, loc, \sigma\}$: Tuples contain information on the conference name and location. The value σ reflects the importance of the conference in its field.

```
SELECT TOP 100 C.conf
FROM Research R, Travel T, Conference C,
     People P, Research R1, Travel T1
WHERE ((R.conf=C.conf)
   or (R.person=T.person and T.loc=C.loc)
   or (R.person=P.person
       and ((P.advisor=T1.person and T1.loc=C.loc)
            or (P.advisor=R1.person and R1.conf=C.conf))))
   and R.person IN PREDEF-SET
```

Figure 2: Query retrieving top 100 conferences that researchers in PREDEF-SET are likely to attend, based on factors F1–F4.

Note that in our model we assume, as in other prior work [9, 1], that the scores of tuples in each table are available. Such scores may be computed based on surveys (e.g., $Conference.\sigma$); by machine learning methods (e.g., examine historical attendance records to learn a model for $Research.\sigma$); or by formulas provided by the query issuer (e.g., the marketer believes that $People.\sigma$ should be computed as $(years)^{-1}$, where $years$ is the number of years since a person's graduation; if table $People$ contains attribute $years$ instead of σ, then σ is computed on the fly). A full discussion on modeling tuple scores is beyond the scope of this paper. If all tables were stored in a single DBMS, the marketer would issue the SQL query in Figure 2.

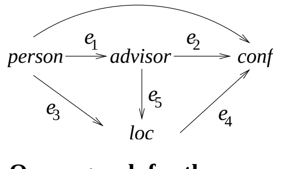


Figure 3: Query graph for the example query

Query graphs It is easier to visualize this SQL query as the query graph in Figure 3. Each edge corresponds to a table, while each node corresponds to an attribute[2]. If two edges share a node,

[2]We restrict the model to binary tables. Tables with more join attributes can be modeled as multiple binary tables.

then there is a join on that attribute between the two tables. For example, edge e_6 corresponds to table $Research$, and edge e_3 to table $Travel$. Edge e_5 also corresponds to $Travel$. The reason we represent this table by two edges is that the table appears twice in the query, as T and $T1$.

Nodes connected by a path correspond to a logical 'and' between their corresponding joins. Thus, the path $person$ - loc - $conf$ corresponds to the clause *R.person=T.person and T.loc=C.loc*. Edges emanating from the same node correspond to a logical 'or' between the clauses that start with the corresponding tables. Thus, since edges e_6 and e_3 start two paths from the same node, the corresponding clauses *(R.conf=C.conf)* and *(R.person=T.person and T.loc=C.loc)* are connected by 'or'.

We use directions on the edges to ensure that certain paths are impossible. For example, the path $person$ - loc - $advisor$ - $conf$ would be a valid path in an undirected graph. However, this would correspond to a clause *(R.person=T.person and T.loc=T1.loc and T1.person=R1.person and R1.conf=C1.conf)* being 'or'-connected to the other conditions. Such a clause breaks the semantics of the SQL query: for $R.person = A$, $T.loc = Shanghai$, and $C.conf = ICDE09$, there are many values $T1.person = B$ that satisfy this clause, because there are many other researchers that are connected to $ICDE09$. However, this should not contribute to the likelihood that A will attend $ICDE09$. To insure the equivalence between the query semantics and the paths in the query graph, we impose directions on edges. Nevertheless, our methods are directly applicable to undirected graphs, as well.

Finally, in order to fix the source and destination nodes, we use the techniques proposed in [14]. The source attributes are the ones that have selection conditions in the "WHERE" clause, and the destination attributes are the ones that appear on the "SELECT" clause. For instance, the example query above has $person$ in the "WHERE" clause with selection condition and $conf$ in the "SELECT" clause, therefore we fix them as source and destination nodes respectively. For simplicity, we assume there are exactly one source and one destination (otherwise, add new nodes s and t; connect s to all sources via edges with scores 1; connect all destinations to t via edges with scores 1).

3. DEFINITIONS

We study join queries of type *SELECT $\mathcal{L}$ from $\mathcal{R}$ where $\mathcal{C}$*, where $\mathcal{R}$ is a list of tables, $\mathcal{L}$ is a list of attributes from $\mathcal{R}$, and $\mathcal{C}$ is a set of join conditions over attributes from $\mathcal{R}$, connected by and/or operators. For the remainder of this paper, we assume that the join query is represented as a query graph, as described in the previous section.

Let $G = (V, E)$ be the (directed or undirected) query graph, with *source node* s and *destination node* t; $s, t \in V$. Each edge $e \in E$ corresponds to a table accessible via a Web site, and thus has an associated set of tuples denoted $Tup(e)$. For each tuple τ, let $\sigma(\tau) \in [0, 1]$ denotes the score of τ. Similarly, each node $v \in V$ corresponds to an attribute and has an associated domain denoted $Val(v)$. The domain contains all possible values for that attribute, over all the tables that have that attribute. For any edge e, if its endpoints are nodes u and v, then $Tup(e) \subseteq Val(u) \times Val(v)$.

3.1 Cost Model

Our goal is to minimize the number of Web accesses necessary to compute the query results. As in [7], we consider two types of probes: *random access probes (RA)* and *sorted access probes (SA)*. We first define them below, and then explain their contribution to the cost function.

In an RA probe, we know the value for at least one position in the tuple, and we ask for all the tuples that match that value, along that edge. An SA probe, on the other hand, returns the tuple with highest score that has not been accessed so far. We use the notations $RA(e)$ and $SA(e)$ to denote random and sorted accesses on edge e respectively.

Whenever a tuple τ is returned as part of an RA or SA result, we assume that its score $\sigma(\tau)$ is also returned. An RA probe may return more than one tuple. If k tuples are returned, the cost of the operation is $Cost_{RA} + \alpha(k-1)Cost_{RA}$, where $Cost_{RA}$ is the cost of one Web access, and $0 < \alpha < 1$ is a dampening factor. The rationale is that having a Web request processed by a remote site is the main bottleneck, and the number of results returned adds only a small overhead. By contrast, an SA probe only returns one request at a time. However, since these results are accessed sequentially, it is reasonable to assume that multiple results are sent at once, and cached on the query processor's site. Therefore, we assume that $Cost_{SA} = \beta Cost_{RA}$, for some $0 < \beta < 1$.

3.2 Bindings

We define a *query result* to be a set of tuples, one from each table in the 'FROM' list $\mathcal{R}$, such that the tuples satisfy the conditions in the 'WHERE' clause $\mathcal{C}$. The set of values for the columns in the 'SELECT' list $\mathcal{L}$ can easily be computed from the query result. A brief justification for this definition is provided in Remark 1 at the end of this subsection. This set of tuples induces a binding of all nodes in the graph to some specific values. In addition, it also induces corresponding scores on the edges. Conversely, a binding of nodes to values and edges to scores, if it is consistent with the query conditions, induces a unique query answer (and its score). For the sake of clarity, we therefore refer to query results as *complete bindings*, defined below.

DEFINITION 1. *Let $G = (V, E)$ be a directed query graph, where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. A* complete binding *of G is a vector*

$$B = (a_1, \ldots, a_n, \sigma_1, \ldots, \sigma_m),\ a_i \in Val(v_i)$$

such that, for any edge $e_i = v_j \to v_k$, if the tuple (a_j, a_k) belongs to $Tup(e_i)$ then $\sigma_i = \sigma((a_j, a_k))$; and otherwise, $\sigma_i = 0$. We say that edge e_i is bound to the tuple (a_j, a_k)*, and nodes v_j, resp. v_k, are* bound to the values a_j*, resp. a_k.*

Note that we must allow zero-score values on edges in order to model situations in which not all paths can be instantiated. For example, the vector $(A, SIGPOD09, Providence, B, 0, 0.8, 0.9, 0.4, 0.9, 0.7)$ is a complete binding of the query graph in Figure 3. Tuple $(A, SIGPOD09)$ is not an instance of table e_1. Therefore, $\sigma_1 = 0$. Tuple $(A, Providence)$ is an instance of e_2, with score 0.8.

Our branch-and-bound strategy involves exploring and possibly discarding a subset of complete bindings (i.e., complete results) at each step. We represent such subsets as partial bindings (i.e., partial results), defined below.

DEFINITION 2. *Let $G = (V, E)$ be a query graph, where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. We denote by '*' a new symbol, such that $* \notin (\cup_{i=1}^{n} Val(v_i))$. A* partial binding *of G is the vector*

$$PB = (b_1, \ldots, b_n, [\ell_1, L_1], \ldots, [\ell_m, L_m]),\ b_i \in (Val(v_i) \cup \{*\}),$$

such that for each $1 \leq j \leq m$, $[\ell_i, L_i] \subseteq [0, 1]$ and $[\ell_i, L_i]$ contains at least one score $\sigma(\tau)$ of a tuple $\tau \in Tup(e_i)$.

For any $v_i \in V$, we use $PB[v_i]$ to denote the value of PB corresponding to v_i (i.e., $PB[v_i] = b_i$). Similarly, for any $e_j \in E$, $PB[e_j]$ denotes the range $[\ell_j, L_j]$ corresponding to e_j.

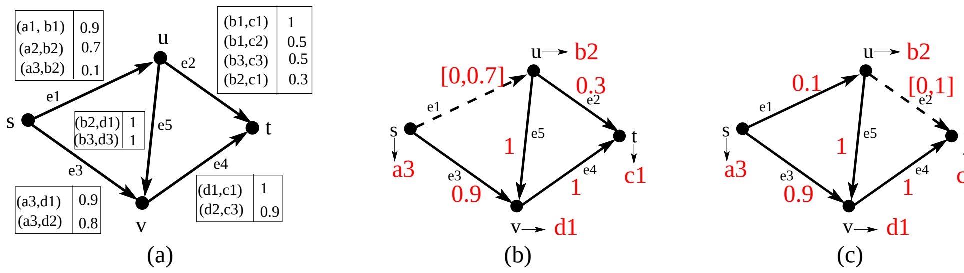


Figure 4: Generating bindings for a simplified version of the graph in Figure 3 (dashed edges are unbound): (a) the graph and its associated edge tuples and scores; (b), (c) two different partial bindings.

Note that, unlike a complete binding, a partial binding allows a node instance b_i to be the new symbol *. This signifies that node v_i has not been bound to any instance from $Val(v_i)$. For the range of an edge e_i, we will only allow two cases: Either $\ell_i = 0 < L_i$, in which case we say that e_i is *unbound*; or $\ell_i = L_i = \sigma(\tau)$, where $\sigma(\tau)$ is the score of a tuple $\tau \in Tup(e_i)$. In the latter case, we say that e_i is *bound* to the tuple τ, and denote it by $e_i \rightarrow \tau$.

As we detail in Section 4, our algorithm generates new partial bindings PB' from a current partial binding PB using probes on unbound edges e_i. In general, in the new partial bindings edge e_i will be bound to one of the tuples $\tau \in Tup(e_i)$ returned by the probe (some exceptions occur for SA probes).

Executing one edge binding: We use the notation $PB' = (PB, e_i \rightarrow \tau)$ to signify that PB' was created from PB by binding edge e_i to τ. Edge e_i must be unbound in PB. More precisely, PB' is computed as follows: $PB'[e_i] = \sigma(\tau)$; if $e_i = v_j \rightarrow v_k$ and $\tau = (a, b)$, then $PB'[v_j] = a$ and $PB'[v_k] = b$; all other entries in PB' are the same as in PB. This edge binding operation is well-defined only if τ is *compatible with* PB, i.e., $PB[v_j] \in \{a, *\}$ and $PB[v_k] \in \{b, *\}$. In other words, we only execute an edge binding $e_i \rightarrow \tau$ if the endpoints of e_i are either unbound, or bound to the same values as in τ.

EXAMPLE 2: *Consider the query graph from Figure 4(a). A complete binding for this graph is, e.g.,*

$$B = (a_3, b_2, c_1, d_1, 0.1, 0.3, 0.9, 1, 1).$$

Two partial bindings for the graph are illustrated in Figures 4(b) and (c): unbound edges are dashed, while bound ones are solid; ranges/scores are indicated along the edges; and the binding values for nodes are indicated by small arrows. Hence, Figure 4(b) illustrates the partial binding

$$PB_1 = (a_3, b_2, c_1, d_1, [0, 0.7], 0.3, 0.9, 1, 1),$$

and Figure 4(c) corresponds to

$$PB_2 = (a_3, b_2, c_1, d_1, 0.1, [0, 1], 0.9, 1, 1).$$

Note that, even though the nodes are bound to the same values in all 3 cases, the bindings are different, because they were generated via different edge bindings. For example, $B = (PB_1, e_1 \rightarrow (a_3, b_2)) = (PB_2, e_2 \rightarrow (b_2, c_1))$, but PB_1 and PB_2 cannot be generated from each other via edge bindings.

An example of invalid edge binding in this figure is $(PB_1, e_1 \rightarrow (a_2, b_2))$, since it conflicts with the binding of node s to a_3 in PB_1.

Intuitively, a partial binding is a short-hand notation for a subset of complete bindings. It is therefore natural to talk about an inclusion relationship between bindings, as follows.

DEFINITION 3. *Let PB_1 and PB_2 denote two partial bindings, such that*

$$PB_1 = (b_1, \ldots, b_n, [\ell_1, L_1], \ldots, [\ell_m, L_m])$$

$$PB_2 = (c_1, \ldots, c_n, [r_1, R_1], \ldots, [r_m, R_m]).$$

We say that PB_1 is included in PB_2, and write $PB_1 \subseteq PB_2$, if for all $1 \le i \le n$, either $c_i = b_i$ or $c_i = *$; and for all $1 \le j \le m$, $[\ell_i, L_i] \subseteq [r_i, R_i]$. If, in particular, PB_1 is a complete binding and is included in PB_2, we say that PB_1 belongs to PB_2 and write $PB_1 \in PB_2$.

REMARK 1. *In the example from Section 2, there is a unique complete binding for each pair $(R.person, C.conf)$. However, this is not usually the case. Suppose that table $Travel$ has an extra attribute $OptionID$, and that it contains tuples t_1 and t_2 as $(ID1, A, Providence, 0.9)$ and $(ID2, A, Providence, 0.88)$. Then the answer $(A, SIGPOD09)$ is obtained via 2 complete bindings B_1 and B_2: B_1 binds edge e_2 to t_1 with a score of 0.9, while B_2 binds it to t_2 with a score of 0.88. Returning B_1 and B_2 as separate results gives the marketer additional information; e.g., he may have airline clients interested in it. Moreover, our algorithms can still be adapted to return just $(A, SIGPOD09)$, with score $score(B_1)$, i.e., the maximum score of all complete bindings generating the pair.*

3.3 Computing Scores of Bindings

Let $G = (V, E)$ be a query graph with specified *source node* s and *destination node* t; $s, t \in V$. Graph G can be seen as a communication network, in which s transmits a signal that t must receive. The signal can travel along any edge. An edge $e \in E$ fails (gets disconnected) with probability $1 - \pi(e)$, where $\pi(e)$ is the *success probability* of e. The probabilities of different edges are assumed to be independent. The probability that a path $P = e_1 e_2 \ldots e_k$ succeeds, i.e., that the signal travels from one end to the other of P, is therefore $\pi(P) = \Pi_{i=1}^{k} \pi(e_i)$. The *reliability* of network G is the probability that at least one of the paths between s and t succeeds; equivalently, it is the probability that G remains connected. Given the equivalence between the boolean conditions in a SQL query Q, and the structure of its corresponding query graph G, we propose scoring the answer to Q as the network reliability of G. More precisely,

DEFINITION 4. *Let $B = (a_1, \ldots, a_n, \sigma_1, \ldots, \sigma_m)$ be a complete binding of G. For any edge e_i, we define its success probability as $\pi(e_i) = \sigma_i$ (recall that $\sigma_i \in [0, 1]$). We define the score of B, denoted $score(B)$, to be the reliability of network G under these edge probabilities.*

For partial bindings

$$PB = (b_1, \ldots, b_n, [\ell_1, L_1], \ldots, [\ell_m, L_m]),$$

we will compute a range of scores $[min(PB), max(PB)]$ as follows: Let the *minimum network of* PB, resp. *maximum network of* PB, be the network G where the success probability of any edge e_i

is defined as $\pi(e_i) = \ell_i$, resp. $\pi(e_i) = L_i$. Then $min(PB)$, resp. $max(PB)$, is the reliability of the minimum, resp. maximum, network of PB. The following result will be used in Section 4 to explain our strategy for choosing edge probes.

PROPOSITION 1. *Let PB_1, PB_2 be two partial bindings such that $PB_1 \subseteq PB_2$. Then:*

(i) $[min(PB_1), max(PB_1)] \subseteq [min(PB_2), max(PB_2)]$. In particular, if PB_1 is a complete bindings, then $score(PB_1) \in [min(PB_2), max(PB_2)]$.

(ii) If there exists at least one path P such that all edges of P are bound to non-zero values in PB_1, but at least one such edge is unbound in PB_2, then $min(PB_1) > min(PB_2)$. If no such path exists, then $min(PB_1) = min(PB_2)$.

PROOF. See Appendix A. □

Computing the reliability of a general network is NP-Hard [13]. The Monte-Carlo algorithm in [4] approximates the reliability of a network with arbitrarily high precision. Multiple iterations are executed, and the precision increases with the number of iterations. Note that one could also compute the network reliability in a deterministic way by the inclusion/exclusion formula over paths. However, the complexity of this approach grows exponentially with the number of paths, and quickly becomes impractical. Therefore, we will employ the Monte-Carlo algorithm for computing the scores of bindings, and assume that enough iterations are executed so that all approximation errors are negligible.

4. TOP-K ALGORITHM

In this section, we present our algorithm for efficiently computing the top-k complete bindings of a query graph. Our cost model assumes that tuple scores are stored remotely and are expensive to access. To this end, we design an efficient edge probing strategy that computes the top-k bindings based on a subset of tuple scores.

Our strategy generalizes Fagin's Threshold Algorithm (TA) [2]. The TA algorithm assumes that each object in a database has m attributes stored in m lists. The score of an object is computed using some monotonic aggregation function f, such as min or average. The algorithm works by doing sorted access in parallel to each of the m sorted lists. For each object B that is seen under sorted access, TA then does a random access to the other lists to find the corresponding scores for object B and computes its overall score $f(B)$. Only the k objects with highest overall score are stored, at any given time. TA defines the *threshold value* τ to be $f(\underline{x_1}, \ldots, \underline{x_m})$ (where $\underline{x_i}$ is the last object seen under sorted access on list i) and halts when the k highest scores are at least equal to τ.

In our setting, the objects correspond to complete bindings, and the m attributes of an object B correspond to the m edge bindings in B. The value of an attribute is the score of the corresponding edge binding. The monotonic function f is $score(B)$. However, a direct application of the TA algorithm is impossible in our model, as we explain below. Suppose we started by doing a sorted access in parallel on all edges, i.e., an SA probe on each edge. For each binding $e_i \rightarrow SA(e_i)$ that is retrieved under sorted access, we would need to know the object B to which it belongs. However, in our case, one edge binding may be part of many complete bindings, and we have no way of identifying them at this point. Even if an edge binding occurred in only one complete binding B for which we could somehow obtain an identifier, the TA algorithm would still require random accesses on all other edges (using B's id) to find all the edge bindings in B and their scores. Clearly, this would lead to many expensive edge probes.

Instead, our approach modifies the TA method in several crucial ways: We maintain sets of objects together, and compute lower and upper bounds for the scores of all objects in a set. Each such set has a succint representation as a partial binding. We may store more than k (complete or partial) bindings at any given point. While we still do sorted access in parallel over all edges, we do not follow such a step by compulsory RA probes on all edges. Instead, we design and study several strategies for deciding what RA probes to execute.

Throughout this section, we use the query graph from Figure 4(a) to illustrate these ideas. This graph is obtained from the query graph in Figure 3, where edge e_6 was removed for simplicity. As mentioned above, we assume that each edge in the graph has a sorted list of tuple scores, in descending order of scores. Ties are broken in an arbitrary but fixed manner. We say that the topmost tuple has level 1, the next tuple has level 2, a.s.o. We will maintain a global level s, which is originally set to 0, i.e., the pointer in each sorted list lies above the first tuple. To execute SA probes in parallel on all edges, we increment s and access the tuple at level s on each edge. If an edge has fewer than s levels, then the result of its SA probe is undefined, and no further SA probes are executed.

$PB_{*,0} = (*, *, *, *, [0, 1], [0, 1], [0, 1], [0, 1], [0, 1])$
$PB_{*,1} = (*, *, *, *, [0, 0.9], [0, 1], [0, 0.9], [0, 1], [0, 1])$
$PB_{*,2} = (*, *, *, *, [0, 0.7], [0, 0.5], [0, 0.8], [0, 0.9], [0, 1])$
$PB_{*,3} = undefined$

Table 1: AllStar bindings for the graph in Figure 4(a).

Our algorithm employs parallel SA probes to generate bindings in which all nodes and edges are unbound, but edge ranges are progressively tighter. We call such bindings *AllStar*. More precisely, the *AllStar of level* s is defined as $PB_{*,s} = (*, \ldots, *, [0, \sigma_1^s], \ldots, [0, \sigma_m^s])$, where σ_i^s is the score of the tuple on level s in the sorted list of e_i. For $s = 0$, $PB_{*,0} = (*, \ldots, *, [0, 1], \ldots, [0, 1])$.

EXAMPLE 3: *The graph in Figure 4(a) has AllStar bindings of levels 0, 1, and 2. They are depicted in Table 1.*

Our overall approach is described in Algorithm 1 shown in Appendix B. It takes as input a query graph G, which comprises, in addition to its node and edge structure, information about the data sources from which edge tuples can be retrieved (via edge probes).

The algorithm maintains a set of partial bindings $\mathcal{S}$, and a set of complete bindings $\mathcal{T}$. Initially, $\mathcal{S} = \{PB_{*,0}, PB^1, \ldots, PB^k\}$, where PB^i is the partial binding having the source node bound to the ith value in PREDEF-SET, and all other nodes and edges unbound; and $\mathcal{T} = \emptyset$. As the algorithm executes the *while* loop, partial bindings from $\mathcal{S}$ are replaced by new bindings with fewer unbound edges. Eventually, some of the partial bindings in $\mathcal{S}$ become complete bindings, and may be added to $\mathcal{T}$. The set $\mathcal{T}$ stores at most k complete bindings at any given time, and they are the bindings with highest scores. The algoritm terminates when $|\mathcal{T}| = k$. It may also terminate sooner if $\mathcal{S}$ becomes empty, which occurs if the query graph has fewer than k complete bindings (Step 25).

During each iteration, we select the binding PB' with maximum upper bound $max(PB')$. If PB' is a complete binding, we add it to $\mathcal{T}$. Otherwise, PB' is replaced with one or more bindings PB'' such that $PB'' \subseteq PB'$ (when adding PB'' to $\mathcal{S}$, we also compute $[min(PB''), max(PB'')]$). Each such computation requires either a round of parallel SA probes, or an RA probe, depending on whether or not PB' is AllStar. We explain each case below.

Replacing an AllStar (Steps 9-14): We first increment the level s and execute all SA probes in parallel, as explained above. If at least one probe is undefined, then we do not generate any new bindings. In this case, no subsequent iteration will enter Step 10 (note that PB' is deleted from $\mathcal{S}$ in Step 6). If, however, all probes are valid, we add the new AllStar to $\mathcal{S}$. We also bind each edge e_i in turn to its tuple of level s, i.e., to $SA(e_i, s)$. In total, we add exactly $|E| + 1$ new partial bindings in Steps 12 and 13. It is trivial to verify that all these new bindings are included in PB'. We make the observation that the set $\mathcal{S}$ contains exactly one AllStar as long as the algorithm passes the test in Step 11, and no AllStar thereafter.

EXAMPLE 4: *Table 2 shows three of the six bindings added to $\mathcal{S}$ during the first iteration, as a result of selecting $PB' = PB_{*,0}$ in Step 5. Refer also to the graph in Figure 4(a).*

REMARK 2. *In Step 13 of Algorithm 1, we could also take an "eager" approach, by creating partial bindings in which several compatible edges are simultaneously bound. We show that such an approach is actually more inefficient than the "lazy" approach we employed. See Appendix C for details.*

Replacing other bindings (Steps 15-23): For ease of presentation, we have omitted some details in Step 16 of Algorithm 1. More precisely, the edge e chosen in this step must have at least one of its endpoints bound to a value, since otherwise we cannot execute an RA probe. Suppose that $e = u \to v$. If both u and v are bound to values a, resp. b, then the RA probe asks whether the tuple (a, b) exists on edge e. If it does, then e is bound to the score $\sigma((a, b))$; otherwise, e is bound to 0; the bindings of u and v remain the same in either case. If only one endpoint of e is bound, it is possible that the RA probe returns multiple tuples. In that case, we bind e in turn to each such tuple. In general, there are multiple unbound edges with one bound endpoint. We choose one randomly from among them.

The resulting new bindings are added to $\mathcal{S}$, provided that they satisfy the conditions in Step 19. We discuss the second condition first. Clearly, this condition ensures that we keep $\mathcal{S}$ as small as possible, and that we do not run unnecessary iterations by selecting duplicate bindings in Step 5. Moreover, it also ensures that we do not double-count complete bindings in the result set $\mathcal{T}$. The test can be executed very efficiently by keeping a hash table on the bindings in $\mathcal{S}$. The next example illustrates how duplicates may arise.

EXAMPLE 5: *Consider two different iterations over the graph from Figure 4(a): In the first iterations, we choose $PB' = PB_1$ in Step 5, while in the other iteration, we choose $PB' = PB_2$ in Step 5; PB_1 and PB_2 are the bindings defined in Table 2. Suppose that for PB_1, we choose the edge $e = e_2$ in Step 16, and for PB_2 we choose $e = e_1$ in Step 16. Table 3 shows the bindings generated during Steps 15-23 of each iteration. Since PB_3 is generated as a duplicate during the second iteration, it is not added to $\mathcal{S}$ again.*

We now discuss the first condition in Step 19. Recall that we wish to generate new bindings PB'' from PB' such that $PB'' \subseteq PB'$. The test $\sigma(\tau) \leq L(e)$ ensures this for all bindings generated in Step 20. The following example illustrates a situation when the test fails, i.e., $\sigma(\tau) > L(e)$.

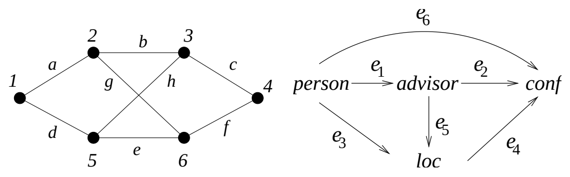
EXAMPLE 6: *Consider the iteration over the graph from Figure 4(a), in which Step 5 chooses $PB' = PB_5$ as depicted in Table 4. (Binding PB_5 was added to $\mathcal{S}$ in Step 13 of an earlier iteration, since $PB_5 = (PB_{*,2}, e_2 \to (b_1, c_2))$.) Suppose that for PB_5, we choose the edge $e = e_1$ in Step 16. Then $L(e_1) = 0.7$, since the range for e_1 is $PB_5[e_1] = [0, 0.7]$. The RA probe $RA(e_1, u \to b_1)$ returns the tuple (a_1, b_1), with score $0.9 > 0.7$. Therefore, binding PB_6 is not added to $\mathcal{S}$. Note that $PB_6 \subseteq PB_4$, where $PB_4 \in \mathcal{S}$ is defined as in Table 3. Hence, all complete bindings contained in PB_6 are also contained in PB_4, and we do not miss any information by ignoring PB_6. On the contrary, we eliminate a redundant partial binding.*

To prove that Algorithm 1 works correctly we need the following two lemmas.

LEMMA 1. *Let B be a complete binding added to $\mathcal{T}$ in some iteration i. Then $score(B) \geq max(PB)$ for any partial binding PB that belongs to $\mathcal{S}$ at the end of any iteration j, $j \geq i$.*

$PB' = PB_5 = (*, b_1, c_2, *, [0, 0.7], 0.5, [0, 0.8], [0, 0.9], [0, 1])$
$PB_6 = (PB_5, e_1 \to (a_1, b_1))$: $(a_1, b_1, c_2, *, 0.9, 0.5, [0, 0.8], [0, 0.9], [0, 1])$

Table 4: Enforcing the inclusion property for the graph in Figure 4(a): $PB_6 \not\subseteq PB_5$, so PB_6 is not added to $\mathcal{S}$.



(a) Synthetic join graph (b) Real-life query graph

Figure 5: Graphs used in experiments

LEMMA 2. *Let B be a complete binding that is never added to $\mathcal{T}$. Then at the end of each iteration in Algorithm 1, there exists at least one binding $PB \in \mathcal{S}$ such that $B \in PB$ (and therefore, $score(B) \in [min(PB), max(PB)]$).*

PROOF. See Appendix D ☐

Let B be a complete binding. We claim that if $B \notin \mathcal{T}$ at the end of Algorithm 1, then all complete bindings of $\mathcal{T}$ have scores larger or equal to $score(B)$. Let $B' \in \mathcal{T}$ be an arbitrary complete binding. Since $B \notin \mathcal{T}$, Lemma 2 implies that after the *last* iteration, $\mathcal{S}$ contains a partial binding PB such that $B \in PB$. Therefore, $score(B) \leq max(PB)$. By Lemma 1, $max(PB) \leq score(B')$. Hence, $score(B) \leq score(B')$, and this is true for any $B' \in \mathcal{T}$. We conclude with the following.

THEOREM 1. *For any query graph G that admits at least k complete bindings, the set $\mathcal{T}$ returned by algorithm top-$k(G)$ contains the top-k complete bindings of G.*

5. EXPERIMENTAL EVALUATION

In this section we report the results of the extensive experimental study we conducted to evaluate the benefits of our approach for various query graphs and data distributions. We implemented our method using Java with SDK 1.5 and ran experiments on a CentOS machine with 3.0 GHz Intel Xeon CPU and 16 GB RAM.

5.1 Experiment setup

We implemented Algorithm 1, which throughout this section is referred as the SMART method. In all experiments, PREDEF-SET is the entire domain of the source attribute. We also implement a rank-join [3] based approach (RJ) as follows: The rank-join algorithm is first applied to each join path to generate the top-k join results with the scoring function being the product of all edge scores. We then apply the rank-join algorithm to the graph treating each path as data sources to produce the overall top-k join results with the scoring function being the network reliability. Note that we extend the original rank-join algorithm to consider random access as well as sorted access. We do not compare with the naive approach which instantiates and sorts all join results because both approaches we study are orders of magnitude better.

We consider various graphs in our experiments. We evaluate our approach using both synthetic and real world datasets (the motivating example) as detailed in Appendix E. Due to space constraints, we show experimental results only for one synthetic join graph (see Figure 5(a)), and for the join graph over real world

$PB_{*,1} = (*, *, *, *, [0, 0.9], [0, 1], [0, 0.9], [0, 1], [0, 1])$
$PB_1 = (PB_{*,1}, e_1 \to (a_1, b_1)) = (a_1, b_1, *, *, 0.9, [0, 1], [0, 0.9], [0, 1], [0, 1])$
$PB_2 = (PB_{*,1}, e_2 \to (b_1, c_1)) = (*, b_1, c_1, *, [0, 0.9], 1, [0, 0.9], [0, 1], [0, 1])$

Table 2: Bindings computed during the first iteration for the graph in Figure 4(a).

Step 5:	$PB' = PB_1 = (a_1, b_1, *, *, 0.9, [0, 1], [0, 0.9], [0, 1], [0, 1])$
Steps 15-23:	$PB_3 = (a_1, b_1, c_1, *, 0.9, 1, [0, 0.9], [0, 1], [0, 1])$
	$PB_4 = (a_1, b_1, c_2, *, 0.9, 0.5, [0, 0.9], [0, 1], [0, 1])$
Step 5:	$PB' = PB_2 = (*, b_1, c_1, *, [0, 0.9], 1, [0, 0.9], [0, 1], [0, 1])$
Fails Step 19:	$PB_3 = (a_1, b_1, c_1, *, 0.9, 1, [0, 0.9], [0, 1], [0, 1])$

Table 3: Bindings generated in Steps 15-23 of two different iterations, for the graph in Figure 4(a): PB_3 is generated twice, but only added once to $\mathcal{S}$.

datasets from Figure 5(b). For synthetic datasets, we consider various types of data distribution (uniform v.s. skewed, uncorrelated v.s. correlated). We evaluate the performance by counting the number of SA and RA probes, as defined in Section 3.1. We set α=0.1 and β=0.1 and report $Join\ Cost\ = \sum_{RA\ probe} Cost_{RA} + \sum_{SA\ probe} Cost_{SA}$.

5.2 Uniform Datasets

Figure 6(a) shows the Join Cost for the SMART and RJ methods for the uniform uncorrelated dataset. The x-axis is the number of top-k answers computed. We vary k from 10 to 100. As shown, the SMART method clearly outperforms the RJ method in all four distributions. In addition, the cost of the RJ method is the same over all k values. This can be explained as follows. First of all, since multiple paths may share the same edge and the RJ method is applied to each path of the graph, it incurs cost on the same edge repeatedly (*e.g.*, path $a \to b \to c$ and $a \to g \to f$ share edge a). More importantly, the RJ method computes the top-k result on the path level, making it difficult to decrease the *threshold* value [3]. Assume RJ joins path p_1 and p_2 and it computes the threshold value as $max(f(E^{(1)}_{top}, E^{(2)}_{current}), f(E^{(1)}_{current}, E^{(2)}_{top}))$, where $E^{(i)}_{top}$ and $E^{(i)}_{current}$ refer to the edge scores of the top-1 and current join result on path p_i, and f is the computation of network reliability. Even if the current join result on path p_i has a low score, it could still have high scores on a few edges along the path, making the score of the overall join result high. In fact, we observe in the experiments that even the top-1 join query requires the RJ method to retrieve all join results on each path, which explains why the RJ method has the same cost over all k values. Compared with the RJ method, the SMART method reduces the cost by 68% on average.

5.3 Skewed and Correlated Datasets

Figure 6(b), 6(c) and 6(d) show the performance comparison for skewed and correlated datasets. As shown, the performance gain of the SMART method magnifies as the datasets have skewed and correlated distribution. The RJ method performs similarly over skewed, correlated, and uniform datasets, largely due to the fact that it has to instantiate all the join results on each path. By contrast, the SMART method performs better over the skewed (32%) and correlated dataset (24%), versus the uniform dataset. We attribute this cost reduction to the fact that in the skewed dataset the tuple scores drop faster, and thus the SA probes could effectively reduce the upper bound of unseen bindings. For the correlated dataset, our SMART method benefits by identifying early a few partial bindings instantiated from the correlated path edges that are likely to have very high scores.

5.4 Real-World Experiments

We show in Appendix E how we extract real world datasets. Table 5 shows the top-3 bindings as well as the edge scores for the real dataset experiment. As shown, our algorithm returns reasonable results for such a real life query. In particular, all edges are instantiated for each of the 3 bindings, indicating that every path contributes to the final score of the bindings. Although the third binding has the highest score on one of the paths (the single edge path e_1), the other two bindings have relatively high scores on all paths, therefore and result in higher overall score.

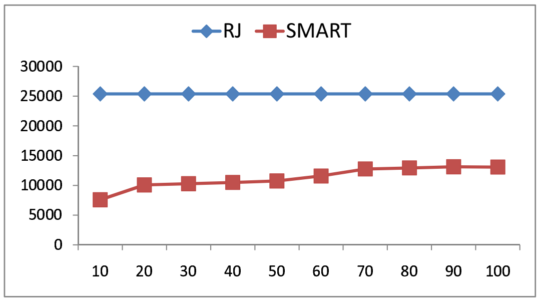
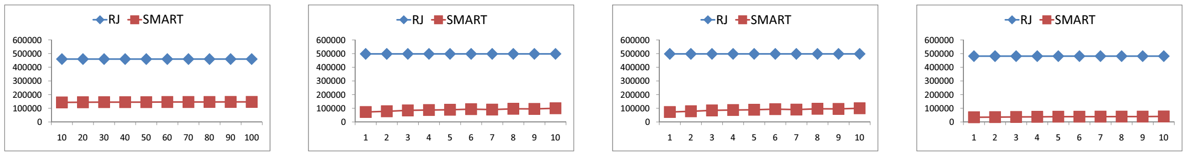


Figure 7: Cost of top-k join queries for the real-world dataset

Figure 7 shows the cost of the SMART and RJ approaches for the real-world experiments. Similar as the synthetic experiments, the SMART method achieves significant cost savings compared with the RJ method. On average, the SMART method beats the RJ as much as 70%. This demonstrate that our algorithm is practical when used in real life applications.

6. RELATED WORK

Top-k query processing has been studied extensively in various areas; see, e.g., [2, 7, 12]). In the typical top-k query model, the score of each object is computed based on a number of attributes stored at data sources. The best known top-k algorithm is the threshold algorithm (TA) proposed by Fagin et al. in [2], which requires both sorted and random accesses. The NRA algorithm improves over TA by considering only sorted access, which is cheaper than random access. Marian et al. [7] proposed the Upper strategy for the case when only random access is available. Theobald et al. [12] studied top-k queries with probabilistic guarantees and proposed a series of approximate variants of TA to reduce the run-time cost. However, all these studies assume that a universal ID for each object is available in each data source, which is not practical in our join query scenario. In fact, under the join model in our work, an object - which in this case is a complete binding - is only known after the scores on all data sources are probed.



(a) (b) (c) (d)

Figure 6: Cost of top-k join queries for synthetic join graph: (a) Uniform Uncorrelated; (b) Skewed Uncorrelated; (c) Uniform Correlated; (d) Skewed Correlated

Bindings	$(e_1, e_2, e_3, e_4, e_5, e_6)$	score
(*"Tao Li", "Washington", "Mitsunori Ogihara", "CIKM 2004"*)	(0.833, 0.34, 0.75, 0.8, 0.833, 0.34)	0.9627
(*"Tao Li", "Toronto", "Mitsunori Ogihara", "SIGIR 2003"*)	(0.667, 0.7, 0.8, 0.8, 0.667, 0.7)	0.9471
(*"Daphne Koller", "Seattle", "Joseph Y. Halpern", "IJCAI 2001"*)	(0.9, 0.142, 0.9, 0.003, 0.9, 0.041)	0.9137

Table 5: Top-3 Bindings of real-world experiments

Algorithms for top-k join query processing have been proposed in [3, 8]). Ilyas et al. [3] introduced a rank-join algorithm that makes use of the individual orders of its inputs to produce join results ordered on a user-defined scoring function. The rank-join algorithm [3] outperforms the $J*$ algorithm [8] by using a score-guided join strategy, effectively reducing the score threshold. However, as we mentioned in Section 1, these two approaches are designed for a single join path and cannot be directly applied to the join graph considered in this paper. In addition, both of their models consider inner join, assuming that each answer in the top-k set meets the join condition and instantiate scores on each data source, whereas in our join graph model, a binding could instantiate a subset of the data sources and still have a high score.

A complete binding of the query graph can be translated into a DNF formula, with one clause corresponding to each source-to-sink path. In this context, Ré et al. [9] proposed a novel approach for top-k queries in probabilistic databases. The method runs several Luby-Karp simulations [4] in parallel, to approximate the score for each answer. However, their approach requires that all answers be computed a priori, and the goal is to minimize the number of simulations. In our model, pre-computing all answers means accessing all scores in each remote data source, which simplifies to the `Naive` approach. In fact, our explicit goal is to minimize the number of such source probes. Note, though, that the two approaches are orthogonal: one could combine them in order to minimize both probing and computation costs.

Top-k query processing in probabilistic database is studied in [10, 6, 15]. In probabilistic databases, the rank of an item is decided by its score in combination with its probability. Soliman et al. [10] investigate two top-k semantics (U-Top$k$ and U-kRanks) in uncertain databases and propose new formulations for top-k queries. Yi et al. [15] propose an improved version of algorithms for the same query. Li et al. [6] propose two parameterized ranking functions (PRF^{ω} and PRF^{e}) for top-k query in probabilistic databases and present novel generating function-based algorithms for efficient query processing.

Theobald et al. [11] design the TopX retrieval engine for the top-k query processing for semistructured data. In their work, they adopt the *eager* strategy to join tuples obtained from sources after a round of sorted access, which as we have discussed (Remark 2) could be incorrect. In addition, TopX assumes that there exists a unique ID for each document (*doc id*) and it is accessible from each tuple, which makes it not directly applicable to our problem. As such, the *eager* strategy is limited to join tuples from sources that are neighbors of each other.

7. CONCLUSIONS

We proposed a novel branch-and-bound approach for top-k join query processing, under a cost model in which data access is expensive. Each data instance has an associated score. We model the score of the overall answer as a network reliability problem. Our algorithm dynamically retrieves a subset of the data on each join edge, and maintains tight upper and lower bounds for sets of answers. We conduct experiments with different types of datasets and query graphs, and show that our algorithm significantly outperforms the rank-join algorithm. The benefits further improve if data scores are correlated and/or skewed, which is often the case for real-life datasets.

8. REFERENCES

[1] N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *PODS*, 2007.

[2] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(1), 2003.

[3] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[4] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *FOCS*, 1984.

[5] B. Kimelfeld and Y. Sagiv. Maximally joining probabilistic data. In *PODS*, pages 303–312, 2007.

[6] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. In *VLDB*, pages 502–513, 2009.

[7] A. Marian, L. Gravano, and N. Bruno. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2), 2004.

[8] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.

[9] C. Ré, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.

[10] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.

[11] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. Topx: efficient and versatile top- query processing for semistructured data. *VLDB J.*, 17(1):81–115, 2008.

[12] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, 2004.

[13] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8:410–421, 1979.

[14] X. Yang, C. M. Procopiuc, and D. Srivastava. Recommending join queries via query log analysis. In *ICDE*, pages 964–975, 2009.

[15] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases. In *ICDE*, pages 1406–1408, 2008.

[16] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley (Reading MA), 1949.

APPENDIX

A. PROOF OF PROPOSITION 1

PROOF. Since $PB_1 \subseteq PB_2$, claim (i) is immediate.

To see why (ii) is true, suppose first that there exists a path P satisfying the conditions as stated. Let $e_i \in P$ be an edge which is unbound in PB_2. This implies that $PB_2[e_i] = [0, L_i]$, so in the minimum network of PB_2, $\pi(e_i) = 0$. Therefore, path P always fails, so it contributes nothing to the network reliability $min(PB_2)$. By contrast, since $PB_1[e_i] = \sigma_i > 0$ for all edges $e_i \in P$, it follows that $\pi(P) > 0$ in the minimum network of PB_1, so P contributes towards the network reliability $min(PB_1)$. Because $PB_1 \subseteq PB_2$, any other path P' has at least the same probability in the minimum network of PB_1 as in the minimum network of PB_2. This implies that $min(PB_1) > min(PB_2)$. For the last claim, if no path P satisfies the stated conditions, it follows that: either P contains an unbound edge in both PB_1 and PB_2; or all edges of P are bound in both PB_1 and PB_2. In the first situation, $\pi(P) = 0$ in both minimum networks, while in the second situation, $\pi(P)$ is the same in both minimum networks. Since this is true for all paths P, $min(PB_1) = min(PB_2)$. □

B. TOP-K ALGORITHM

Algorithm 1 Finding top-k Complete Bindings

```
top-$k(G)$[H]
 1: $\mathcal{S} \leftarrow \{PB_{*,0}, PB^1, \ldots, PB^k\}$
    {where $PB^i = (Val_i, *, \ldots, *, [0,1], \ldots, [0,1])$}
    {$Val_i$: $i$th value in PREDEF-SET}
 2: $\mathcal{T} \leftarrow \emptyset$
 3: $s \leftarrow 0$ {level of SA probes}
 4: while $|\mathcal{T}| < k$ do
 5:   pick $PB' \in \mathcal{S}$ s.t. $max(PB') = \max_{PB \in \mathcal{S}} max(PB)$
 6:   delete $PB'$ from $\mathcal{S}$
 7:   if $PB'$ is complete binding then
 8:     $\mathcal{T} \leftarrow \mathcal{T} \cup \{PB'\}$
 9:   else if $PB'$ is AllStar then
10:     $s \leftarrow s + 1$; do SA probes of level $s$ on all edges
11:     if all SA probes are defined then
12:       $\mathcal{S} \leftarrow \mathcal{S} \cup \{PB_{*,s}\}$
13:       $\mathcal{S} \leftarrow \mathcal{S} \cup \{(PB_{*,s}, e_i \rightarrow SA(e_i, s))\}, \forall e_i$ : edge
14:     end if
15:   else
16:     choose unbound edge $e$: $PB'[e] = [0, L(e)]$
17:     do RA probe on $e$
18:     for each tuple $\tau \in RA(e)$ do
19:       if $\sigma(\tau) \leq L(e)$ AND $(PB', e \rightarrow \tau) \notin \mathcal{S}$ then
20:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{(PB', e \rightarrow \tau)\}$
21:       end if
22:     end for
23:   end if
24:   if $\mathcal{S} == \emptyset$ then
25:     return $\mathcal{T}$
26:   end if
27: end while
28: return $\mathcal{T}$
```

C. EAGER VS. LAZY APPROACHES

In Step 13 of Algorithm 1, we could also take an "eager" approach, by attempting to create partial bindings in which several compatible edges are simultaneously bound. In Example 4, such a binding could be $PB_{1,2} = (PB_{*,1}, e_1 \rightarrow (a_1, b_1), e_2 \rightarrow (b_1, c_1))$, which is valid, since both edge bindings require the value in node u to be b_1. Instead, we ignore this possibility, and allow the algorithm to generate $PB_{1,2}$ in Step 20 of a later iteration, either as $(PB_1, e_2 \rightarrow (b_1, c_1))$, or as $(PB_2, e_1 \rightarrow (a_1, b_1))$. Suppose that $PB_{1,2}$ is generated as $(PB_1, e_2 \rightarrow (b_1, c_1))$, during the iteration for which PB_1 is chosen in Step 5. This will require executing the RA probe $RA(e_2, u \rightarrow b_1)$ in Step 17 of that iteration. Hence, we will access the tuple (b_1, c_1) for a second time (the first time was as the result of the probe $SA(e_2, 1)$.) Therefore, we appear to be inefficient when it comes to minimizing the number of edge probes.

There are two reasons for which we choose this "lazy" approach to edge binding in Step 13. First, notice that executing the RA probe $RA(e_2, u \rightarrow b_1)$ in a subsequent iteration is not superfluous, as this probe also returns the tuple (b_1, c_2), which is not returned by the probe $SA(e_2, 1)$. In fact, if after the first iteration $\mathcal{S}$ contained only $PB_{1,2}$, but not PB_1, then we could not later generate any complete bindings in which $e_1 \rightarrow (a_1, b_1)$ and $e_2 \rightarrow (b_1, c_2)$. But discarding such complete bindings at this point is incorrect, as we cannot guarantee that they are not among the top-k. The correct alternative is to put both $PB_{1,2}$ and PB_1 in $\mathcal{S}$, thus increasing the size of $\mathcal{S}$. This is a non-trivial problem: In the extreme case, all $|E|$ edge bindings $e_i \rightarrow SA(e_i, 1)$ may be mutually compatible (instead of just e_1 and e_2). In such a case, the eager approach would have to add $2^{|E|}$ partial bindings to $\mathcal{S}$ in order to maintain correctness (each of these bindings would leave a different subset of edges unbound).

Second, note that if $PB_1 \in \mathcal{S}$, it may still be selected in Step 5 of a later iteration, which may still trigger the RA probe $RA(e_2, u \rightarrow b_1)$. We conclude that the lazy approach is in fact more efficient than the eager one.

D. PROOF OF LEMMA 1 AND 2

PROOF. *Lemma 1*: We use induction on iteration j. For $j = i$: B is added to $\mathcal{T}$ if and only if B is selected in Step 5, so $score(B) = max(B) \geq max(PB)$ for any PB that belongs to $\mathcal{S}$ in iteration i. Suppose the claim is true for some iteration j. In iteration $j + 1$, the only new partial bindings PB'' in $\mathcal{S}$ are those generated either in Steps 9-14, or in Steps 15-23, from the binding PB' chosen in Step 5. As discussed above, $PB'' \subseteq PB'$, which implies $max(PB'') \leq max(PB')$. Since PB' belongs to $\mathcal{S}$ after iteration j, $max(PB') \leq score(B)$, and the claim follows. □

PROOF. *Lemma 2:* Each edge e_i in B is bound to a tuple $\tau_i \in Tup(e_i)$, with tuples on adjacent edges having compatible node bindings. Let s_i be the level of tuple τ_i in the sorted list on edge e_i. Without loss of generality, assume that $s_1 \leq \ldots \leq s_m$. Then along each edge e_i, any tupple on a level $s \leq s_1 - 1$ has score at least as large as $\sigma(\tau_i)$. We deduce that $B \in PB_{*,s}$ for all $s \leq s_1 - 1$. Moreover, the algorithm passes the test in Step 11 during any iteration prior to choosing $PB' = PB_{*,s_1-1}$ in Step 5. Therefore, $\mathcal{S}$ contains one $PB_{*,s}$, with $s \leq s_1 - 1$, during all such iterations, (If the algorithm returns without ever choosing PB_{*,s_1-1} in Step 5, then our claim holds).

Once PB_{*,s_1-1} is chosen in Step 5, Steps 9-14 are executed. The test in Step 11 is still true, since there exist tuples τ_i at levels $s_i \geq s_1$ on all edges e_i. Therefore, $PB_1 = (PB_{*,s_1}, e_1 \rightarrow \tau_1)$ is added to $\mathcal{S}$. Note that PB_1 binds edge e_1 to tuple τ_1, the same as B. For all $i \geq 2$, $PB_1[e_i] = [0, L_{s_1}(e_i)]$, where $L_{s_1}(e_i)$ is the score of the tuple on level s_1 in e_i. Since τ_i has level $s_i \geq s_1$, it follows that $\sigma(\tau_i) \in [0, L_{s_1}(e_i)]$. We deduce that $B \in PB_1$.

The binding PB_1 remains in $\mathcal{S}$ until PB_1 is chosen in Step 5 of a later iteration. Then, Steps 15-23 are executed. Let e_k denote the edge chosen in Step 16; e_k must be adjacent to e_1, so we can do an RA probe. Since τ_k is compatible with τ_1, tuple τ_k is among those returned by the RA probe. Moreover, $\sigma(\tau_k) \leq L_{s_1}(e_k)$, as discussed above. Therefore, $PB_2 = (PB_{*,s_1}, e_1 \rightarrow \tau_1, e_k \rightarrow \tau_k)$ is added to $\mathcal{S}$, and $B \in PB_2$. We can now repeat this argument with PB_2 instead of PB_1. By induction, we show that after any iteration there exists $PB_r \in \mathcal{S}$ with r bound edges, $r \leq m$, such that $B \in PB_r$. If $r = m$ and $PB_m = B$ is added to $\mathcal{S}$, then it is never deleted, since B is never selected in $\mathcal{T}$. $\square$

E. GRAPHS AND DATASETS USED IN EXPERIMENTS

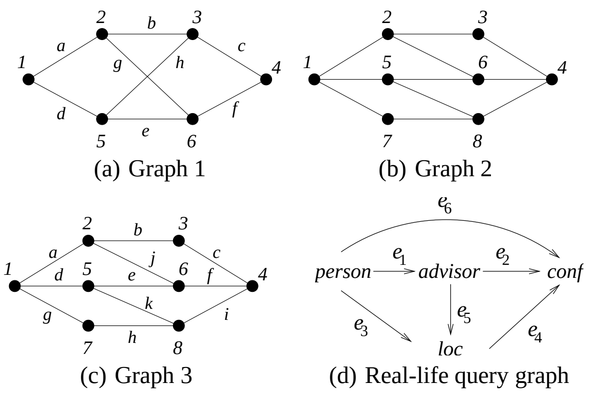


Figure 8: Graphs used in experiments

For testing purposes, we created three different graphs, in order to study the effect of various graph properties on the efficiency of each method. Figure 8 shows the three graphs used in our experiments (with numbers annotating nodes and letters annotating edges). Because of limitation of space, we only present results for Graph 1. In all three graphs, we assume the leftmost and the rightmost nodes are the source and destination nodes, respectively. Rather than assigning directions to edges in some arbitrary manner, we choose to use undirected edges. This is because the number of undirected paths between the source and destination is higher than the number of directed paths, making each instance more challenging. We want to point out, however, that our methods are directly applicable to both directed and undirected graphs.

Graph 1 has 8 distinct paths between the source and destination nodes, such as $a-b-c$ and $a-g-e-h-c$. It also has 9 minimum cuts; for instance, (a, d) or (c, h, d). Graphs 2 and 3 are designed to have comparable number of nodes and edges as Graph 1, but to significantly differ in either the number of paths, or the number of cuts. Recall that, by Proposition 1(ii), the lower bound on the score of a binding increases only when an entire new path is bound. In graphs with more paths, we expect the lower bounds to increase more quickly. Conversely, one can see that the upper bound decreases quickly if we bind an edge that belongs to many cuts. We expect this to happen for Graph 3, We list the number of nodes, edges, paths and cuts of the three graphs in Table 6. Graph 1 has similar number of cuts and paths; Graph 2 has more cuts than paths; while Graph 3 has more paths than cuts.

We test our algorithm using both synthetic and real world dataset.

[Synthetic Dataset]: We generate a variety of datasets for our experiments, which model different types of real-life instances. For each edge in one of the three graphs, we must generate tuples and

	Nodes	Edges	Paths	Cuts
Graph 1	6	8	8	9
Graph 2	8	11	9	27
Graph 3	7	10	16	8

Table 6: Graph Statistics

their corresponding scores. Let (v_i, v_j, $score$) denote a scored tuple, where v_i and v_j represent the values of the tuple corresponding to the end nodes of its edge, and $score$ is its score. Each tuple may join with multiple tuples on other edges. In our dataset, we set the number of tuples on each edge to 200 and the average fan-out of each tuple to 4. The tuple scores are generated randomly, as explained below. We are interested in studying the effect of the following two parameters on the efficiency of the methods:

- **Uniform vs. Skewed score distribution** We generate two datasets: In the first dataset, scores on an edge are drawn from the uniform distribution on $[0, 1]$. In the second dataset, scores on an edge follow the Zipf's distribution [16]. With a traditional Zipf's distribution (s = 1), the tuple score is the inverse of its rank.
- **Edge-Correlated vs. Uncorrelated scores** Tuples that join, from adjacent edges, may or may not have correlated scores. We test the performance of our approach in both scenarios. For correlated datasets, we pick a join path for which a high-score tuple from one edge implies high scores of the join partners from other edges. We limit the correlations to be among the top few (10%) tuples on the selected path.

[Real-world Dataset]: We use the motivating example discussed in Section 2 for the real-world experiment (Figure 8(d)). In such a query, we are trying to find the top-k bindings (*person, location, advisor, conference*). In particular, edge scores are computed as follows.

- The scores of edges e_1 and e_5 are computed based on the researcher's papers accepted by the conference. For each paper, the researcher gets a score of 1 divided by the number of authors of the paper. For example, a researcher gets a score of 0.7 if he has two papers with 2 and 5 authors accepted by the conference. Since this score can reach a value greater than 1, we set an upper bound of 0.9.
- The scores of edge e_2 and e_6 are computed as 100 divided by the distance (in miles) between the researcher and the conference location, with an upper bound of 0.7.
- We assign a score between 0.3 to 0.9 to edge e_3 based on the conference reputation.
- The relation score between a researcher and his or her advisor (edge e_4) is based on the graduation year: it gets a score of 0.8 when the researcher was still under supervision and decrease by a factor of 2 every year after graduation.

We extracted data from a snapshot of the DBLife dataset, which contains the publication and conference information up to the year of 2006. In order to find genealogy information of researchers, we use the data from the AI Genealogy Project[3], which provides genealogy information for researchers in AI area. By corroborating the data from AI Genealogy and DBLife, we were able to check out 59 AI researchers, as well as their advisors. We manually retrieved

[3]http://aigp.eecs.umich.edu/

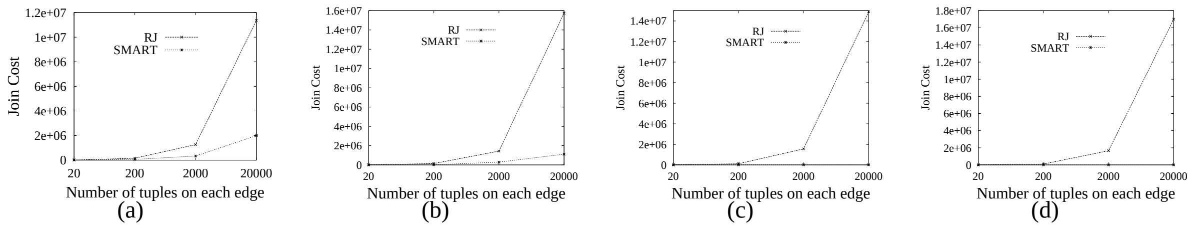


Figure 9: Cost of top-k join queries for the large scale dataset: (a) Uniform uncorrelated; (b) Uniform correlated; (c) Skewed uncorrelated; (d) Skewed correlated

the affiliation of the researchers and conference locations and computed the distance between researchers and conferences for edge e_2 and e_6. Our real world dataset[4] contains information for 91 researchers and 110 conferences.

F. LARGE DATASETS

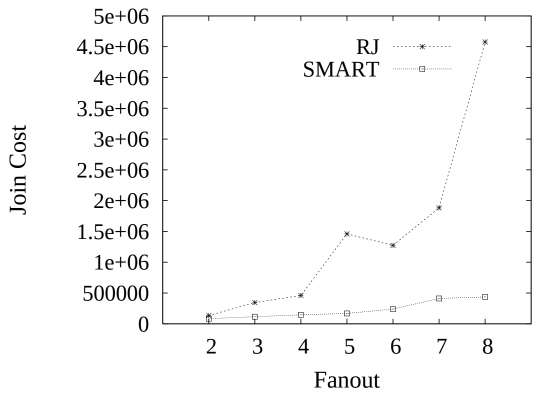


Figure 10: Cost of top-k join queries as a function of fanout

In previous section, we show the performance of our approaches in a fairly small dataset, with each edge hosting 200 tuples with a fanout of 4. Despite the superiority of the SMART method, we are also interested in how it scales in large dataset and with large fanout value. Figure 10 plots the cost for the RJ and SMART methods over uniform uncorrelated dataset for a fanout value from 2 to 8. As the fanout value grows, the cost for the SMART grows steadily in a slow pace. Compared with a fanout of 2, the cost for SMART method grows to 5.2 times for a fanout of 8. On the contrary, the cost for the RJ grows more than 34 times for the same fanout change. These results demonstrate that our approach is extremely suitable in datasets where heavy joins are expected (*i.e.*, large fanout).

Figure 9 plots the cost for the RJ and the SMART methods in large datasets, with the number of tuples on each edge ranging from 20 to 20000 and a fixed fanout of 4. Under four different types of datasets, the SMART method unanimously demonstrates further benefits compared the other two methods. By increasing the size of the dataset by 1000 times, the cost of the SMART method only grows 117.53, 137.33, 2.22, 3.05 times for each of the four datasets respectively. This is because the SMART method can prune out a large set of unnecessary binding processing by maintaining a tight upper bound. On the contrary, the cost of the RJ method grows by a factor of 1084 times among all the four datasets simply because it has to expand all partial bindings on each join path.

[4]http://paul.rutgers.edu/∼alexng/dataset.txt