Sum-Max Monotonic Ranked Joins for Evaluating Top-K Twig Queries on Weighted Data Graphs*

Yan Qi
Arizona State Univ.
Tempe, AZ 85287
yan.qi@asu.edu

K. Selçuk Candan
Arizona State Univ.
Tempe, AZ 85287
candan@asu.edu

Maria Luisa Sapino †
University of Torino
Torino, Italy
mlsapino@di.unito.it

ABSTRACT

In many applications, the underlying data (the web, an XML document, or a relational database) can be seen as a graph. These graphs may be enriched with weights, associated with the nodes and edges of the graph, denoting application specific desirability/penalty assessments, such as *popularity*, *trust*, or *cost*. A particular challenge when considering such weights in query processing is that results need to be ranked accordingly. Answering keyword-based queries on weighted graphs is shown to be computationally expensive. In this paper, we first show that answering queries with further structure imposed on them remains NP-hard. We next show that, while the query evaluation task can be viewed in terms of *ranked structural-joins* along query axes, the monotonicity property, necessary for ranked join algorithms, is violated. Consequently, traditional ranked join algorithms are not directly applicable. Thus, we establish an alternative, *sum-max monotonicity* property and show how to leverage this for developing a self-punctuating, horizon-based ranked join (HR-Join) operator for ranked twig-query execution on data graphs. We experimentally show the effectiveness of the proposed evaluation schemes and the HR-join operator for merging ranked sub-results under sum-max monotonicity.

1. INTRODUCTION

Many types of data, including the web, XML documents or relational databases, can be modeled as graphs. For instance, in a relational database, each node can represent a tuple in the database and links may represent the foreign key relationships. Discover [26] and DBXplorer [3] are examples of systems, built on top of relational databases, that view the underlying relational database as a graph. In many applications, these graphs are also enriched by weights denoting application specific desirability/penalty measures, such as *popularity*, *trust*, or *cost*. For example, in FICSR [38], the data graph represents an integrated set of (potentially) conflicting data models and edge weights represent the amount of *dis*agreement between these models (Figure 1).

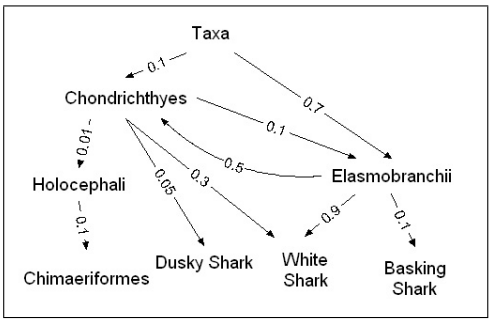


Figure 1: An example weighted data graph segment from FICSR [38]; the weight of a given edge denotes the amount disagreement on the corresponding assertion in a given set of taxonomies to be integrated

When data have weights, not all results are equally desirable: results need to be ranked according to the underlying cost model. For instance, [23] presents an XML query language extended with IR-related features, including weighting and ranking. XRank [25] and ObjectRank [6] compute PageRank [8] style ranking results for keyword-based (IR-style) database queries. XSEarch [17], a search engine for XML data, relies on extended information retrieval techniques for ranking. *Retrieval by information unit* (RIU) [34], BANKS-I [7], BANKS-II[30], and DPBF [19] recognize that in many cases a single node is not sufficient to answer user queries. Instead, given a query consisting of a set of keywords, they try to find *small* subtrees (in a given weighted graph) containing all the query keywords. Finding minimal trees answering keyword queries on weighted graphs is computationally expensive [34]. Since users are usually interested in not all but top-k results, [34, 7, 30, 19] rely on efficient heuristics and approximations for progressively identifying the smallest k trees covering the given keywords.

While answering keyword-based queries on graph data is useful in various application domains, structural relationships between the data elements are also important and may need to be considered along with query keywords or tags [39]. In XML databases, for example, query processors are designed to exploit the tree-like structure of the XML data. In fact, many exist-

*Supported by NSF Grant "*Archaeological Data Integration for the Study of Long-Term Human and Social Dynamics (0624341)*"

†This work was done while the author was at ASU on sabbatical leave.

VLDB '07, September 23-28, 2007, Vienna, Austria.

ing (binary or holistic) structural join operators, including TwigStack/PathStack [9], iTwigJoin [14], and Stack-Tree-Desc/Anc [4], are structurally-informed variants of the standard sort-merge join algorithm: they require that the data nodes are available in a *structurally sorted* order before the join operation can be performed. To implement structural join operations efficiently, most XML query processors rely on index structures based on structurally-informed node labeling schemes (such as Dietz's labeling [18], which assigns interval-labels to nodes in such a way that descendant nodes have intervals that are contained within the intervals of their ancestors). This enables checking the ancestor/descendant relationships quickly. Such structural labeling and sorting are especially feasible when the underlying data has a tree-structure, but becomes non-trivial when the queries have to be evaluated on graph-data.

When the results need to be ranked based on the edge weights of the matches, the *weight*-order of the (sub)matches becomes as important as their structural ordering. In the literature, there are a number of ranked-join algorithms for top-k queries [21, 13, 12, 22, 32]. These rely on weight-sorted input streams for pruning unpromising matches. In particular, [22] presents an NRA algorithm which (a) considers data sources which can provide results only in (progressively) descending order of desirability and which (b) enumerates top-k desirable join results (but does not compute output grades, which would be needed for combining joins in a complex query plan) without having to access all the data from these sources. A common assumption behind all these algorithms, including [22], is that the function which evaluates the score of combined results is monotonic. Yet, as we see in Section 3, in the case of evaluating twig queries on weighted data graphs, a *non-redundancy* property of answers makes such a monotonicity assumption impossible.

1.1 Contributions of This Paper

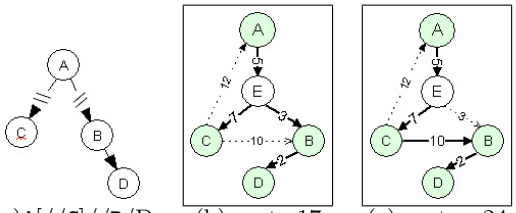
In this paper, we develop **top-k twig query evaluation algorithms for weighted data graphs**. In particular, we

- introduce the top-k twig query evaluation problem on weighted data graphs, present a cost model for the query answers, and prove that answering twig queries on weighted graphs is NP-hard (Section 2);
- show that, while this can be viewed as *ranked structural-joins* along query axes, the monotonicity property, necessary for ranked-join algorithms [21, 22, 13, 12, 27, 33], is violated (Section 3);
- establish a *sum-max monotonicity* property and leverage this for a self-punctuating *horizon-based ranked join* (HR-Join) operator (also in Section 3);
- use HR-Join for optimal or sub-optimal top-k twig-query evaluation plans(Sections 3.4 and 4.2); and
- develop algorithms for progressive sub-result enumeration. These generate streams of sub-matches, combined using HR-Joins (Section 4).

In Section 5, we evaluate the effectiveness of the proposed schemes. We provide an overview of the related literature in Section 6 and present our conclusions in Section 7.

2. PROBLEM FORMULATION

In this section, we first formally pose the problem of *ranked twig query evaluation on weighted data graphs*. We then prove that the problem is NP-hard through a reduction from the group Steiner tree problem.



(a)A[//C]//B/D (b) cost=17 (c) cost = 24

Figure 2: An example query twig and two matches on a weighted graph: the first match with weight 17 is more desirable and should be enumerated and ranked before the second result

2.1 Preliminaries

Data model. We use, $G(V, E)$, to denote a node- and edge-labeled directed graph. Furthermore, we use $tag(v)$ to denote the data-label corresponding to the data node $v \in V$ and $cost(e)$ to denote the cost-label for edge $e \in E$.

Query statement. We model queries as twigs (i.e., tree patterns), possibly described in an XML query language, such as XPath [44][1] or XQuery [45]. Tree patterns can be visualized as trees, where nodes correspond to tag-predicates and edges correspond to a parent/child or ancestor/descendant axes (Figure 2(a)). Thus, a given query, q, can be represented in the form of a node- and edge-labeled tree, $T_q(V_q, E_q)$, where $tag\_pred(qv)$ denotes the tag predicate[2] corresponding to the vertex $qv \in V_q$ and $axis\_pred(qe)$ denotes the axis predicate associated with the edge $qe \in E_q$.

Answer to a query. We define *an answer to q over graph G* as follows (Figures 2(a) and (b)):

Definition 1 (Answer to q Over the Graph G). *An answer to query, $q = T_q(V_q, E_q)$, over the data graph, $G(V, E)$, is a pair, $r = \langle \mu_{node}, \mu_{edge} \rangle$, of mappings:*

- *μ_{node} is a mapping from the nodes of the query tree to the nodes of the data graph, such that given $qv \in V_q$ and the corresponding data node, $\mu_{node}(qv)$, $tag(\mu_{node}(qv))$ satisfies $tag\_pred(qv)$.*
- *μ_{edge} is a mapping from the edges of the query tree to* simple paths *in the data graph, such that given $qe = \langle qv_i, qv_j \rangle \in E_q$, the path μ_{qe}, from $\mu_{node}(qv_i)$ to $\mu_{node}(qv_j)$, satisfies $axis\_pred(qe)$. Let $\mathcal{E}$ denote the set of edges used in the answer to q:*

$$\mathcal{E} = \{e \parallel e \in \mu_{edge}(qe) \text{ for a query edge } qe \in E_q\}.$$

$\mathcal{E}$ does not define a cycle. Note that a path consisting of a single edge can satisfy both parent/child and ancestor/descendant axes, while a multi-edge path can satisfy only ancestor/descendant axes. ◇

Cost of an answer. Given the above definition, we define the cost of an answer as follows:

Definition 2 (Cost of An Answer). *Given an answer, $r = \langle \mu_{node}, \mu_{edge} \rangle$, to query, $q = T_q(V_q, E_q)$, over the data graph, $G(V, E)$, the cost, $cost(r)$, of the answer is defined in terms of the costs of the relevant edges in the graph.*

[1]In XPath [44], only one of the query nodes is returned. We are interested in the entire tree as the result. This is analogous to the tree patterns in [5, 28]. XPath semantics can be implemented as a projection on results.

[2]In this paper, we consider tag equality predicates. The approach extends to other value predicates as well.

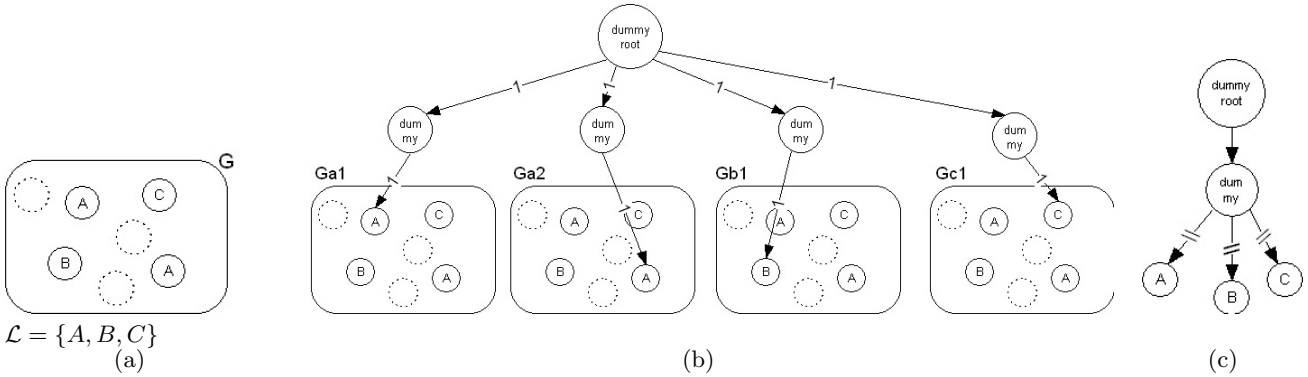


Figure 3: (a) A Steiner problem instance $\langle G, \mathcal{L} \rangle$ consisting of a graph and a set of labels; (b) G' constructed from the instance; (c) the twig query T' constructed from the Steiner problem instance.

Let $\mathcal{E}$ be the set of edges that belong to any of the paths in the answer. The cost is $cost(r) = \sum_{e \in \mathcal{E}} cost(e)$.

Non-redundancy property. When there are overlaps between paths matching query edges, the cost of the common edges are counted only once (Figures 2(b) and (c)). We refer to this as the *non-redundancy* property of the results.

2.2 Top-K Twig Queries over Weighted Data Graphs

Given the definitions of data graph, query statement, answers, and their costs, we formulate a top-k (least-cost) query over G as follows:

DEFINITION 3 (TOP-k QUERY OVER G). *Given a query statement, $q = T_q(V_q, E_q)$, a data graph, G, and a positive integer k, identify (and order in increasing order of cost) a set, R, of k answers such that there are no other answers to q over G cheaper than $max\{cost(r) \parallel r \in R\}$*

If there are less than k answers to q in G, R includes all existing answers (if any). ◇

2.3 Complexity of Top-k Twig Queries

We establish the complexity of evaluating top-k twig queries over graphs by reducing the NP-complete *group Steiner* problem [41] to the min-cost (top-1) twig query problem. Given a node- and edge- labeled graph G, and a set, $\mathcal{L}$, of node labels, let us consider the problem of finding the min-cost tree, t, in G, such that for each label $l_i \in \mathcal{L}$, t contains at least one node labeled l_i. Since, there can be multiple (i.e., a group of) data nodes with the same label, l_i, this is known as the min-cost *group* Steiner tree problem and is known to be NP-complete [41]. In the database literature, the min-cost group Steiner tree problem is used to prove hardness of various problems, such as the *information-unit* [34] and the *min-cost keyword tree* [19] problems.

Let us be given an instance, $\langle G, \mathcal{L} \rangle$ of the min-cost group Steiner tree problem. We reduce this to a corresponding instance of the min-cost twig problem $\langle G', T' \rangle$ as follows:
(Step 1:) Construction of G': For each node, n_i, in G, labeled with $l_j \in \mathcal{L}$, replicate the graph. Let us denote this replica, $G_{i,j}$. For each replica, create a dummy vertex $dv_{i,j}$ with the dummy label, "`dummy`". Connect $dv_{i,j}$ to node n_i in replica $G_{i,j}$. The cost associated to the edge originating from the dummy node is set to 1.

Create a new dummy vertex, dv_0, with the label "`dummyroot`". Connect dv_0 to each of the previously created dummy vertices. The edge cost from dv_0 to the other dummy vertices is set to 1.

The resulting graph is called G'. Figure 3 (b) shows an instance of the construction process. Note that the construction time is polynomial in the size of G and $\mathcal{L}$.
(Step 2:) Construction of T': The corresponding twig query, T', is illustrated in Figure 3 (c). Note that the construction time is polynomial in the size of $\mathcal{L}$.

Given the above construction, an answer to the min-cost twig query is a solution to the min-cost group Steiner tree problem. Any answer to the min-cost twig query will include dv_0 (labeled with "`dummyroot`"), exactly one dummy vertex, $dv_{i,j}$ (labeled with "`dummy`"), and a tree, t, from the subgraph $G_{i,j}$ (identical to G). Due to the construction of T', the tree, t, must include at least one node labeled l_j for each $l_j \in \mathcal{L}$. That is, the tree t is a min-cost group Steiner tree in G for the set $\mathcal{L}$. Since t is contained in an answer to the min-cost twig query, t must be minimal in G.

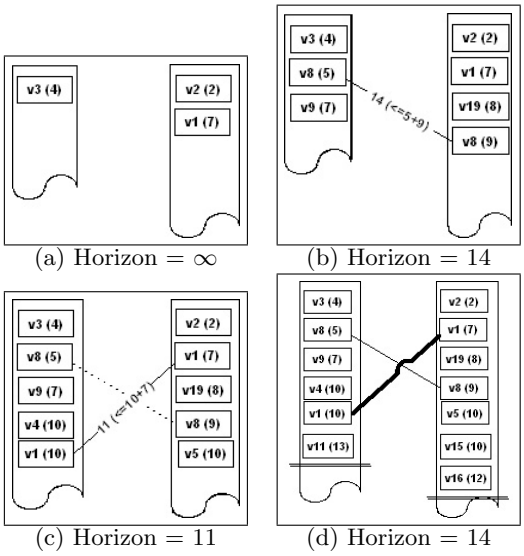
Also, for every solution to the min-cost group Steiner tree problem, there is a corresponding answer to the min-cost twig query. This is true by the construction of G' from G and T' from $\mathcal{L}$. In particular, the replication of the graph ensures that the descendants of the query node "`dummy`" will be constrained into one of the sub-graphs and will, in fact, be a solution to the min-cost group Steiner tree problem.

3. EVALUATING TOP-K TWIG QUERIES ON GRAPHS

Let us consider the query `A[//C]//B/D`, in Figure 2. On XML data, this query can be implemented using structural joins between the data nodes matching the individual query nodes [9, 15, 29] or as a join between data paths satisfying the individual branches [16, 35]. When dealing with top-k queries on weighted data, we face two challenges: not only there can be multiple paths, with different costs, between a given pair of data nodes, but also the underlying join operators need to be rank-aware.

3.1 Sum-Max Monotonicity

Per Definition 2, the cost of an answer is the sum of the costs of the edges involved in the answer, $r = \langle \mu_{node}, \mu_{edge} \rangle$. Since the paths described by μ_{edge} may overlap (i.e., share edges on the graph), the cost of the answer described by $\langle \mu_{node}, \mu_{edge} \rangle$ is not necessarily equal to, but is **bounded by**, the sum of the path costs (Figures 2(b) and (c)). Consequently, due to possible edge overlaps, cost-order of the data paths matching query edges may not correspond to



(a) Horizon = ∞ (b) Horizon = 14

(c) Horizon = 11 (d) Horizon = 14

Figure 4: Ranked join by the *sum-max monotonicity*: (a) no results yet; (b) first candidate with cost 14 is found; (c) a second candidate with lower cost of 11 is found; and (d) stopping *sum-max* condition is reached for returning the current best candidate

the cost-order of the query results.

Existing ranked join algorithms (such as [21, 22, 13, 12, 27, 33]) rely on a *monotonicity* property, which requires that a query plan with better sub-plans is always more desirable. However, since for twig queries sub-results (e.g., data paths matching query axes) are not necessarily independent from each other (i.e., data paths may overlap on edges), the monotonicity property does not hold. Consequently, two costly sub-results with large overlaps may provide a combined result cheaper than two other individually less costly, but non-edge overlapping sub-results.

While the monotonicity condition does not hold, we can establish a range for the costs of query results in terms of the costs of their sub-results.

PROPOSITION 1 (COST RANGE OF ANSWERS). *Let $q = T_q(V_q, E_q)$ be a twig query and let $r = \langle \mu_{node}, \mu_{edge} \rangle$ be a corresponding answer. Let $SR = \{sr_1, sr_2, \ldots, sr_m\}$ be a set of sub-results that give r. Then, the following is true:*

$$\max_{sr_i \in R}(cost(sr_i)) \leq cost(r) \leq \sum_{sr_i \in R} cost(sr_i).$$

This proposition, which follows from the Definitions 1 and 2 (of answer, cost, and the non-redundancy property), enables us to state a *sum-max monotonicity property* for twig results:

PROPERTY 1 (SUM-MAX MONOTONICITY). *Let $q = T_q(V_q, E_q)$ be a twig query and let r_1 and r_2 be two answers. Let R_1 and R_2 be the corresponding sets of sub-results that give r_1 and r_2 respectively. Then, the following is true:*

$$\left(\sum_{sr_i \in R_1} cost(sr_i) \leq \max_{sr_j \in R_2} (cost(sr_j)) \right) \rightarrow cost(r_1) \leq cost(r_2).$$

Next, we leverage this property of twig queries to implement a ranked join algorithm for cost-ordered inputs.

3.2 Progressive Result Enumeration based on the Sum-Max Monotonicity Property

The *sum-max monotonicity* property of answers enables us to leverage the cost evaluations of initial, *candidate*, matches as *horizon*s that limit the candidates that need to be explored before a *confirmed* result can be produced.

EXAMPLE 1. *Let us consider a twig query, q, which consists of two path sub-queries, m_1 and m_2, that join on a query node. Let us also assume that m_1 and m_2 can return paths in cost-order, progressively. Figures 4(a)-(d) show the various stages of the two path streams matching m_1 and m_2, respectively. The individual paths are shown as rectangles, each containing the id of the data vertex (matching the query vertex common in m_1 and m_2) and the total cost of the path. Each stream grows with sub-results arriving in ascending order of cost. The stages of the process are as follows:*

1. *In Figure 4(a), we are seeing a state where none of the sub-results, matching m_1 and m_2, can be joined. At this stage, since there is no join, the upper bound on the cost of the first result is ∞.*
2. *In Figure 4(b), a match is found. The cost of the combined match is* 14. *Note that, although this is the first discovered match, it is not necessarily the best one. Per the* sum-max monotonicity *property (Property 1), this first match sets the horizon for the* best *match to* 14. *Thus, the process has to continue until all the sub-results of cost up to* 14 *are considered.*
3. *In Figure 4(c), a second match, with cost* 11, *is found. Per the* sum-max monotonicity *property (Property 1), this match lowers the* horizon *from* 14 *to* 11. *Thus, the process, now, has to continue only until all the sub-results of cost up to* 11 *are considered.*
4. *In Figure 4(d), the stopping condition is reached: in both sub-result streams, all the paths of cost less than or equal to* 11 *have been considered. Thus, among the two matches found so far, the best (with cost 11) can be returned as the top-1 result.*
5. *When further results are required, the process continues by setting a new horizon. In this example, since there is a known candidate match, the cost (14) of this candidate will be used as the new horizon value.*

Unlike the ranked join algorithms, such as [21, 13, 12], which stop the sorted-access process as soon as k candidates are found, the stopping condition of the above process is based, not on the cardinality of initial candidates but, on their *cost*s. Next we present a horizon based ranked join operator based on the above process.

3.3 HR-Join: Horizon based Ranked Join

In this section, we build a *horizon based ranked join operator (HR-Join)*, based on the progressive, ranked, result enumeration process illustrated above. The HR-Join operator (Figure 5) takes as its input two data streams, in ascending order of data costs, and produces an output data stream (of pairwise joined inputs) in ascending order of combined costs. While internally HR-Join operator uses a standard symmetric non-blocking hash-join, the inputs and outputs to the HR-Join are regulated by (a) two *horizon valves* and (b) a *result sieve*, respectively.

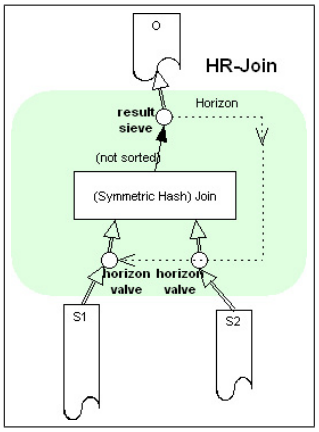


Figure 5: The self-punctuating HR-Join operator consists of a hash-join operator sandwiched between two input regulating horizon valves and an output controlling result sieve. The sieve communicates with the valves through a *horizon* variable; the valves communicate with the sieve through punctuations inserted into the stream

```
horizonValve(ref S, ref horizon)
        /* Controls the flow of cost-ranked data on stream S*/
    /* Punctuates and blocks the stream, S, based on horizon */
begin
  1. local avail=0;
  2. repeat
     (a) if (S.entry[avail + 1] ≠ ⊥)     /* if there is more data*/
         i. if (S.entry[avail + 1] ≠ ∅)   /* if end-of-stream not
            met*/
            A. if (cost(S.entry[avail + 1]) > horizon)
                   S.punctuation = avail+1;
            B. repeat until (S.entry[avail + 1] == ⊥) or
                   (cost(S.entry[avail + 1]) ≤ horizon)
            C. S.punctuation = ∞;
            D. avail++;

     until S.entry[avail] = ∅        /* Until end-of-stream is met*/
end
```

Figure 6: Pseudo-code for the horizon valve: The valve passes the input entries as long as their costs are smaller than the current *horizon*. When the *horizon* is met, the stream is punctuated. This condition persists until the *horizon* value is relaxed[3]

3.3.1 Horizon Valve

Horizon valves control the data availability on a given stream of cost-ranked data. The pseudo-code for the horizon valve module is presented in Figure 6. The inputs to the module are references to a stream, S, and a horizon variable, *horizon*. The externally controlled *horizon* variable is used for regulating the availability of the incoming data. Once the horizon is *met* (i.e., data as costly as horizon is seen in the input stream), the horizon valve punctuates the stream and gets into a waiting state[3]. This state persists until the value of the *horizon* variable is increased by an external process (i.e., the result sieve).

[3]While the pseudo-code in Figure 6 uses busy-wait to implement horizon-based blocking of the stream, the actual implementation relies on more efficient signaling to implement this behavior.

```
resultSieve (ref C, ref horizon, ref O, )
 /* (Incrementally) cost-ranks the candidate stream, C, and creates
a cost-ranked output stream, O*/
  /* Sets the horizon value based on the best candidate available*/
begin
  1. local Top=∅;
  2. local Cand=∅;              /* Implemented as a min-heap*/
  3. local current=0; local numMatches=0;
  4. while (C.entry[current + 1] ≠ ∅) and (numMatches < k) do

     (a) if (C.punctuation > current + 1)
         i. if (C.entry[current + 1] ≠ ⊥)
            A. Cand=Cand ∪ {current + 1};
            B. if (cost(C.entry[current + 1]) <= horizon)
               • Top=current + 1;
               • horizon = cost(Top);
            C. current++;
     (b) else              /* if (C.punctuation == current + 1)*/
         i. numMatches = numMatches + 1
         ii. O.write(Top)   /* Place the confirmed match to O
            in cost-ranked order*/
         iii. Cand=Cand − {Top}
         iv. if Cand ≠ ∅
               • Top=bestinMinHeap(Cand)
               • horizon = cost(Top)
             else
               • horizon = ∞
         v. C.punctuation = ∞

end
```

Figure 8: Pseudo-code for result sieve. The sieve creates a cost-ranked output stream and maintains unqualified candidates in a min-heap. As candidates are identified, the sieve sets the *horizon* value

EXAMPLE 2 (HORIZON VALVE). *Figures 7(a) through (g) depict the operation of the horizon valve on a given cost-ordered stream. Figure 7(a) shows a state where the stream contains three data elements, two of which are already made available to the hash-join. At this state, the value of the horizon variable is ∞ and the stream is not yet punctuated (i.e., the value of the punctuation variable on this stream is ∞). Figure 7(b) shows a state where one more element is made available to the hash-join; since there are no more data, the valve has to wait until a new element or end-of-stream marker is seen. Figure 7(c) shows a state where a new element, with cost* 21*, has arrived. The value of the horizon variable has been set to* 25 *(implying that a join-result with cost* 25 *was generated). In Figure 7(d), the next data entry, with cost* 27*, causes punctuation of the stream. At this stage, the data availability is stopped by the valve until the horizon is increased over* 27 *(Figures 7(e) and (f)).*

This process is repeated until the end-of-stream marker (∅) is reached (Figure 7(g)).

3.3.2 Result Sieve

Since monotonicity does not hold, results produced by the symmetric hash-join are not guaranteed to be cost-ranked. The result sieve module, depicted in Figure 8, takes an unsorted stream of results and (incrementally) creates a cost-ranked output stream.

The process of *sieving* is related to the well-known *heapsort* algorithm, which leverages a *min-heap* data structure to sort a given set of values. Unlike the regular *heapsort*, which requires all the data to be available before the sorted enumeration can start, *result sieve* leverages the sum-max

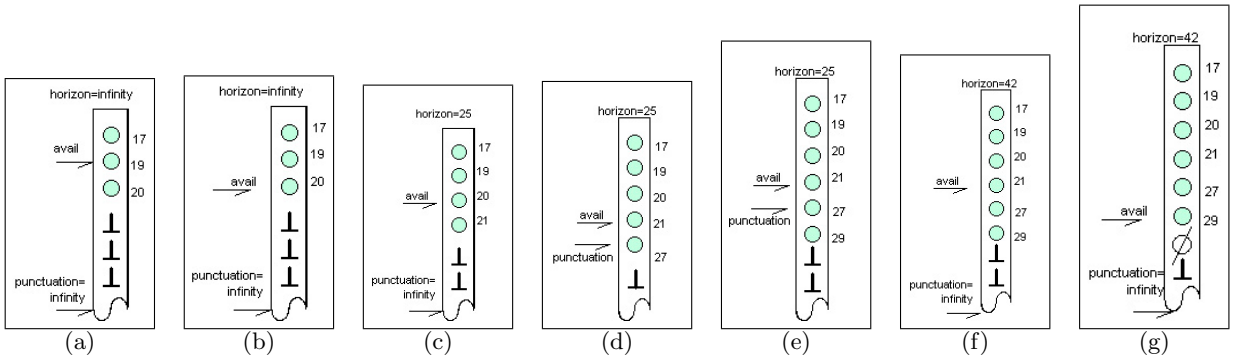
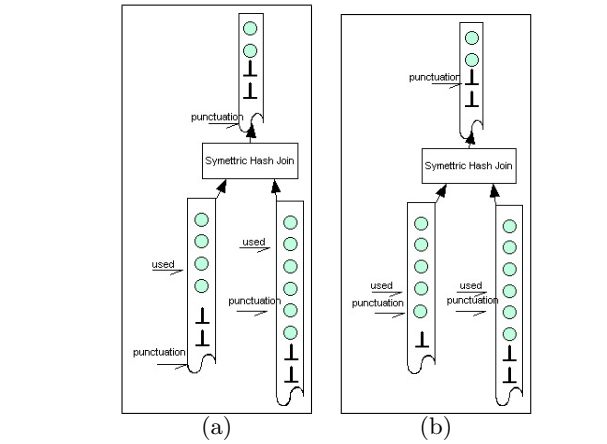


Figure 7: Operation of the horizon valve on a given stream is driven by the externally set *horizon* value. The details are described in Example 2




```
symmetricHashJoin(ref S1, ref S2, ref Sout, qv)
    /* Joins the entries in S1 and S2 on query vertex, qv*/
        /* When S1 and S2 are punctuated, it punctuates the
output stream. */
begin
...
end
```

Figure 9: Punctuation propagation by the symmetric-hash join: (a) Punctuations are not yet met in the input streams, thus the punctuation is set to ∞ in the output stream. (b) When the punctuations are met in the input streams of the symmetric hash-join operator, a punctuation is created in the output stream.

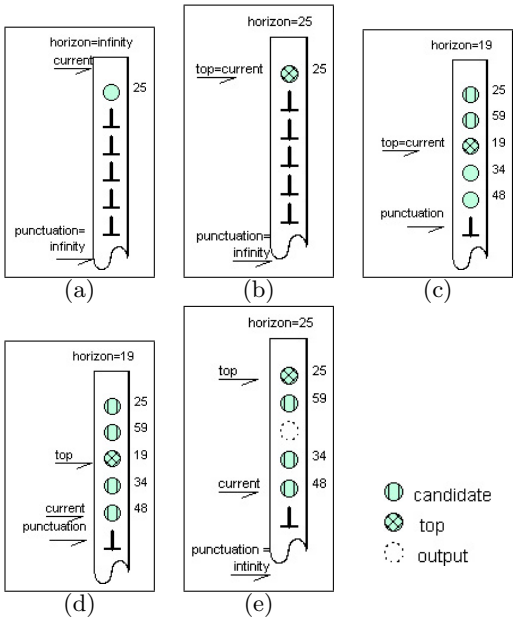


Figure 10: Operation of the result sieve on a given stream of matches produced by the symmetric-hash join. The result sieve *indirectly* regulates its own input stream by updating the *horizon* value. Details are in Example 3

monotonicity property to generate early results. The *result sieve* operator achieves early result enumeration through cooperation with the horizon valves. In particular, horizon valves punctuate the input streams to the join operator based on the *horizon* value set by the result sieve. The join operator propagates these punctuations to the input of the *result sieve*, marking a point suitable for early result enumeration (Figure 9).

Once the current best result is put in the output stream, the *result sieve* pushes the *horizon* value forward, enabling the horizon valves to pass more inputs to the join operator.

EXAMPLE 3 (RESULT SIEVE). *Figures 10(a) through (e) depict the operation of the result sieve on a given stream of matches produced by the symmetric-hash join. Note that the matches passed to the result sieve by the hash join are not necessarily ordered in terms of their costs.*

Figure 10(a) shows the starting state where the horizon value is set to ∞. A match, with cost 25*, is created by the hash join but not processed by the result sieve yet. Figure 10(b) shows the state after result sieve processes this match. Since this is the first match, it is also the best (top) one and the horizon value is set to its cost,* 25*.*

Figure 10(c) shows the state when a better match, with cost 19*, is found. This new match replaces the earlier top and its cost,* 19*, becomes the new, tighter, horizon.*

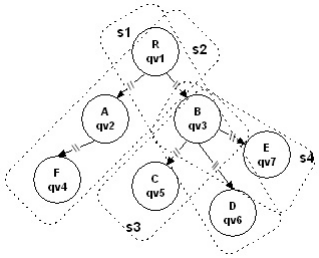
*Figures 10(d) and (e) show what happens when a punctuation is met in the match stream. The top entry when the punctuation is met (Figure 10(d)) is removed from the set of candidates and written to the output stream (Figure 10(e)). The best entry in the remaining set is marked as the new top and its cost (*25*) is used as the new horizon value. Since the current punctuation is consumed, its value is set to ∞.*

This process will be repeated until the next punctuation is seen or until the end-of-stream marker ($\emptyset$) is reached.

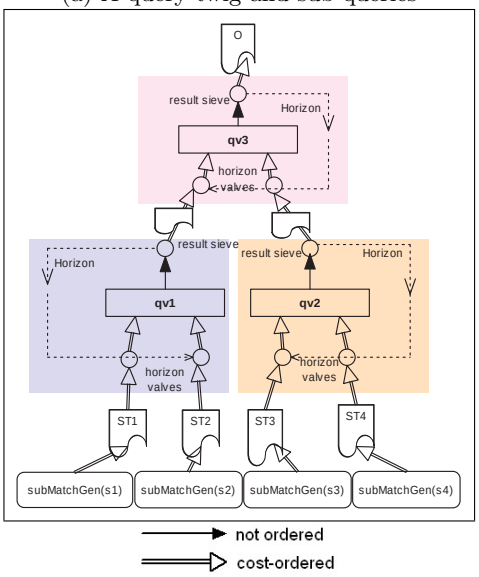
Let us consider m^{th} output of the hash-join. Assuming that

```
subMatchGen(ref S, m, G)
  /* Produces the ranked tree stream, S, for subquery, m*/
begin
   1. local k=0;
   2. repeat
        (a) k ++;
        (b) S.entry[k] := rankedSubQueryExec(m, G, k);
      until S.entry[k] = ∅
end
```

Figure 11: Pseudo-code for generating ranked stream of sub-matches. Details are in Section 4



(a) A query twig and sub-queries



(b) An example query plan using HR-Join operators

Figure 12: (a) A sample query and its sub-queries and (b) an example query plan consisting of four input streams of sub-results (to the sub-queries) and three HR-Join operators

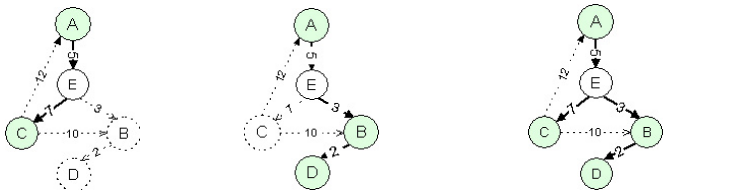
k elements have already been extracted and placed into the output stream, the cost of inserting the new entry to the min-heap is $O(log(m-k))$.

3.4 Ranked Twig Query Processing using HR-Join

Given a query, q, and a set, $\mathcal{S} = \{s_1, \ldots, s_u\}$, of atomic sub-queries, a plan for q conforms to the following outline:

- For each sub-query, $s_i \in \mathcal{S}$, the system generates a cost ranked stream, ST_i, of sub-results (Figure 11).
- The matches for the sub-queries in S are then joined, using HR-Join operators, on the shared query nodes.

Figure 12 shows an example query twig, a possible sub-query partitioning, and a corresponding query plan using three HR-Join operators.



(a)match for A//C (cost = 5+7 = 12) (b)match for A//B//D (cost = 5+3+2 = 10) (c)match for A[//C]//B//D (cost = 5+7+3+2 = 17)

Figure 13: Combination of sub-result within HR-Join operator. Note that the output cost is less than the sum of the input costs due to edge overlaps

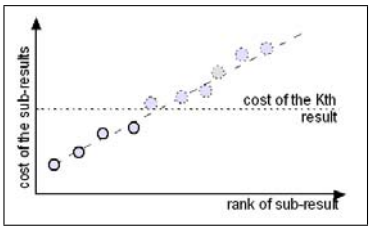


Figure 14: Sub-result pruning by the *sum-max* monotonicity: the sub-results beyond the cost of the k^{th} (overall) result are pruned.

3.4.1 Combining Sub-Results

As described above (and shown in Figure 12), each HR-Join operator in a query plan joins its input stream of sub-results based on the ids of the data nodes matching specific query nodes. Once the hash-join identifies a data node match between two input sub-results, the sub-graphs corresponding to the two sub-results are combined (Figure 13). This is achieved by eliminating redundant nodes and edges (based on the non-redundancy property of answers). The cost of the output graph after combination is equal to the sum of the remaining edges (i.e., less than or equal to the sum of the sub-result costs). Using a data structure which maintains the edges in the sorted order of edgeIDs, the redundant edge elimination step can be implemented using a sort-merge based scheme. Thus, given two sub-results, sr_1 and sr_2, the complexity of this step is $O(|sr_1| + |sr_2|)$.

Results with cycles can be eliminated either *early* (i.e., before insertion into the sieve) or *late* (i.e., when they are candidates for being selected as the *top*) using topological sort in $O(|sr_1| + |sr_2|)$ time.

3.4.2 Sub-Result Pruning

Let us consider a top-k query, q, a set of sub-queries $\mathcal{S} = \{s_1, \ldots, s_u\}$ of q, and the sequence, $a_1, \ldots, a_k$ of k best answers. The *sum-max monotonicity* property (Property 1) implies that, before the k^{th} answer is output by the algorithm, for each sub-query $s_i \in \mathcal{S}$, all the sub-results with costs less than or equal to $cost(a_k)$ need to be considered. Thus, the total number of input sub-matches that *need to be enumerated* for the identification of the k best matches for the twig query is

$$\#necessary\_submatches = \sum_{s_i \in \mathcal{S}} |\{a_{i,j} \parallel (a_{i,j} \in answers(s_i)) \wedge (cost(a_{i,j}) \leq cost(a_k))\}|.$$

The rest of the input sub-results do not need to be considered and can be pruned. As illustrated in Figure 14,

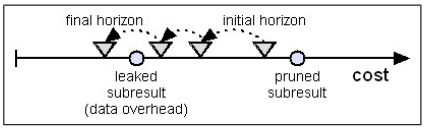


Figure 15: Data overhead: some sub-results may be unnecessarily considered due to an initially lax *horizon* value

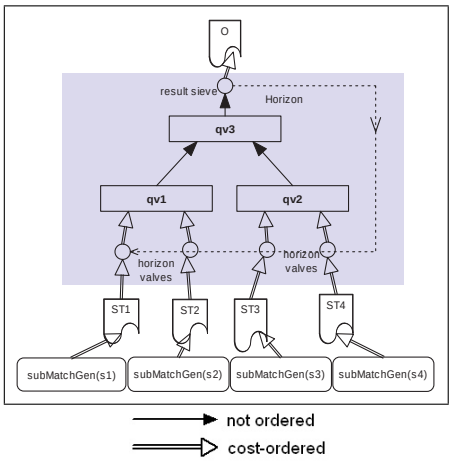


Figure 16: An alternative plan for the sample query in Figure 12. In this plan, there is only one result sieve regulating all the valves of the four input streams. We refer to this as the *M-way* HR-Join

the pruning power of the *sum-max monotonicity* property is determined by the distribution of the costs of the incoming sub-results. Note that, in the worst-case, for a sub-query where all matches cost less than $cost(a_k)$, all of these matches would need to be considered. On the other hand, especially when (a) there are large degrees of overlaps between the sub-results returned by the sub-queries, leading to lower overall costs and (b) when k is relatively low, *sum-max* monotonicity provides significant opportunities for pruning. In Section 5, we evaluate this sub-result pruning property.

3.4.3 Data Overhead

The above equation for the necessary sub-matches relies on the fact that any sub-match beyond the final *horizon* value (i.e., cost of the k^{th} answer) cannot contribute to a cheaper solution. However, as shown in Step 4(a)iB of the result sieve module (Figure 8), *horizon* values are not set once for all, but incrementally tightened as cheaper matches are found. Consequently, there is a possibility that, while the *horizon* is still lax, some sub-matches beyond the tighter, final *horizon* will be passed to the join process (Figure 15). We refer to this as the *data overhead* of the plan:

$$dataoverhead = \frac{\#all\_submatches - \#necessary\_submatches}{\#all\_submatches}.$$

Here $\#all\_submatches$ is the number of sub-matches passed to the join process by any of the sub-queries of q. In Section 5, we evaluate the data overhead of the HR-Join.

3.5 M-way HR-Join

The data overhead discussed in the previous subsection occurs when the *horizon* values set by the result sieves are not sufficiently tight to prevent leakages of unnecessary sub-results to the HR-join operators. One way to tighten the *horizon* values is to use *M-way* HR-Joins, as depicted in Figure 16. In an *M-way* HR-Join, there is only one result sieve regulating all the valves of the M input streams to the join operator. Since the *horizon* values set by the (only) result sieve reflect the costs of the overall results more closely, an *M-way* HR-Join is likely to leak smaller number of unnecessary inputs into the join operator.

The performance of M-way HR-Join based query plans also depends on other parameters. Consider the query plans shown in Figures 12(b) and 16 for the query shown in Figure 12(a): the query plan in Figure 12(b) uses multiple HR-Join operators (each with its horizon valves and result sieve). The plan in Figure 16, on the other hand, has only one global result sieve, controlling all the input streams (to the 4-way join). While, in plan in Figure 12(b) those intermediary results whose cost-orders cannot be confirmed are kept at the intermediary result sieves, in plan Figure 16, all the intermediary results are passed without any check to the upper-most join operator. Consequently, the use of M-way HR-Joins may increase the amount of work needed by the upper join block and the (only) result sieve which needs to maintain all the resulting candidates. In contrast, HR-Join only plans tend to reduce the join work, but increase the cost of the intermediary result sieves. In Section 5, we evaluate the effect of using M-way HR-Join operators in query plans as opposed to using binary HR-Join operators.

4. PROCESSING OF PATH-BASED SUB-QUERIES

Performance of a given query plan also depends on the complexity of its atomic sub-queries. Let us reconsider the example in Figure 12. The example twig query in the figure is split into its sub-queries based on the branches on the twig. While there are many schemes in the XML literature that rely on such path-based query segmentation, data labeling, and indexing [24, 16, 35], in the context of rank-ordered twig query processing on graph data, these schemes are not directly applicable. In particular, different paths between a given pair of data nodes may have different costs and the HR-join operator described in the previous section requires ranked enumeration of input sub-results based on their total edge costs. While, of course, it is possible to pre-compute all paths between all pairs of data nodes and index them for later use, such a scheme would have an exponential-size storage cost. Instead, in this paper, we present algorithms for run-time cost-ranked enumeration of data paths.

4.1 Cost-Ranked Enumeration of Paths

Per Definition 1, an answer, $r = \langle \mu_{node}, \mu_{edge} \rangle$, to query, $q = T_q(V_q, E_q)$, involves a mapping, μ_{edge}, from the edges of the query tree to the *data paths* in the graph. In other words, each query edge/axis can be seen as a path sub-query. Thus, given a query edge, cost-ranked enumeration of best k sub-results matching this edge corresponds to the *k-shortest simple paths* problem, a classical problem in graph theory [46]. Due to its applications in various domains, such as networking, this problem has been studied extensively. Among the alternative solutions, Yen's algorithm for k-shortest paths is preferred due to its general and optimal nature [46, 37]. On a given data graph, $G(V, E)$, this algorithm identifies k shortest loopless paths in G in $O(k|V|(|E| + |V|log|V|))$ time. First the shortest path is obtained; then, this path is deviated to obtain the 2^{nd} shortest path. This is repeated,

one shortest path at a time, until k shortest paths are found. To create the path-based sub-result input streams for HR-Joins, we use a version of Yen's algorithm presented in [37].

Since we do not know in advance the number of sub-results that would be needed for a given sub-query, we run the path-enumeration algorithm to return all sub-results progressively, until instructed to stop. Since Yen's algorithm's enumeration cost is linear in the number (k) of shortest paths returned, the algorithm ensures a relatively constant sub-result inter-arrival time. This is important as it enables regular (and predictable) sub-result arrival to the HR-Join operators and, thus, makes optimization easier.

The inter-result time, $O(|V|(|E| + |V|log|V|))$, is a function of the number of nodes and edges in the data graph. On the other hand, given a pair of data nodes, the only other data nodes and edges that need to be considered in path enumeration are those that are *reachable* from the source node and can *reach* the destination node. While the all-pairs reachability computation is known to be an expensive, $O(V^3)$, operation, this can be done off-line. Thus, given a pair, n_i and n_j, of source/destination nodes, first a reachability graph $G_{i,j}(V_{i,j}, E_{i,j})$, is identified by intersecting the forward and backward reachabilities of n_i and n_j, respectively. Given this, the inter-arrival time of sub-results is

$$interresultTime(n_i, n_j) = O(|V_{i,j}|(|E_{i,j}| + |V_{i,j}|log|V_{i,j}|)),$$

where $V_{i,j}$ and $E_{i,j}$ denote nodes and edges in the corresponding reachability graph segment.

4.2 Horizon Tightening

As we proved in Section 2.3, the problem of min-cost twig query problem on weighted data graphs in NP-Hard. In fact, it is easy to show that, given an arbitrary data graph G, the number of paths between two data nodes can be exponential in the size of the graph. Consequently, the complexity of the HR-Join based twig evaluation plans are hidden in the potential number of path-based inputs.

As discussed in Section 3.4.2, the cost of a HR-Join based plan depends on the number of input paths that *need* be considered before a match can be committed. The HR-join operator uses a *horizon* based self-punctuation mechanism (based on the sum-max monotonicity) to regulate the amount of input paths that need to be considered. Per Proposition 1, the *horizon* value is set assuming that the degree of overlaps between sub-results can be up to 100%. In practice, the degree of overlaps may be significantly smaller than this. To account for this, we introduce a *horizon tightening factor* ($0 < tf < 1$) in Step 2(a)iB of the horizon valve algorithm[4] (Figure 6). An indirect impact of this is a reduction in the inputs to be considered before matches are committed. In Section 5, we evaluate the effect of *tightening factors* on complexity and optimality of the results.

4.3 Tag-based Path Enumeration

Yen's algorithm [37] returns a ranked stream of paths between two nodes in the data graph. However, in twig-query processing, we are mostly interested in paths between pairs of nodes not identified by their node ids, but their tags.

In order to enable tag-based sub-query evaluation, we extend the original graph with two special *start* and *end* nodes per distinct tag in the data (Figure 17). The special *start* nodes are used as sources and the *end* nodes are used as destinations in cost-ranked path enumeration. Since the special nodes do not alter the reachability in the graph and since they do not add any cost to the original paths, they do not affect the order of the matches.

[4] $\ldots (cost(S.entry[avail + 1]) \leq \mathbf{tf} \times horizon)$

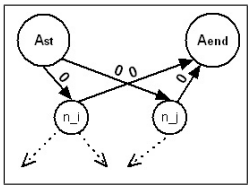


Figure 17: The tag extension of the graph: the two node, n_i and n_j share the same tag, A. Consequently, they are connected to two tag extension nodes labeled with A_{st} and A_{end}, with 0 cost edges

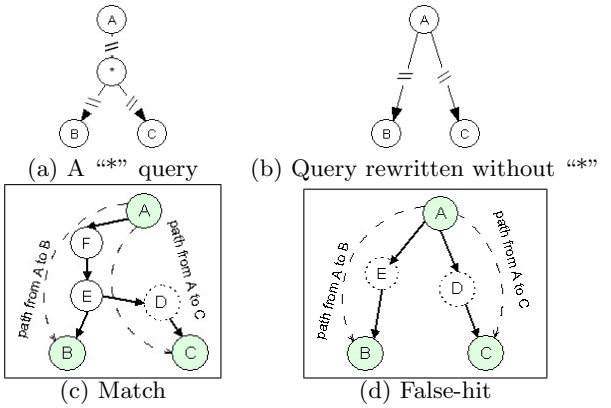
For a given data graph, $G(V, E)$, the overhead of the tag-extension is minimal with $2 \times num\_distinct\_tags$ new nodes and $2 \times |V|$ many new edges.

4.4 Dealing with "*" Wildcards

On one hand, handling of the "*" wildcards, which match all the tags in the data, is trivial through a similar tag extension process. On the other hand, this straightforward solution may prove to be extremely costly. Let us consider a twig query, q=A//*[//B]//C. Splitting this query into three sub-queries, sq_1=A//*, sq_2=*//B, and sq_3=*//C and enumerating all paths in the graph starting from A, all paths ending at B, and all paths ending at C to be later combined using HR-Join operators could be costly.

In order to prevent enumeration of a large number of paths, we refrain from splitting queries at those nodes labeled with "*". Instead, queries are split into sub-queries only at non-wildcard query nodes. For the above example, this leads to two sub-queries instead of three: sq'_1=A//*//B and sq'_2=A//*//C. Paths matching these sub-queries can be identified by cost-ranked enumeration of the data paths from As to Bs and from As to Cs. Given a matching data path, the query node with "*" maps onto any of the intermediary data nodes on the path. Consequently, a single data path of length l, matching the sub-query A//*//B, encodes $l - 2$ sub-results, all with the same edges and, thus, with identical costs. We can generalize this example as follows: Given a sub-query of the form sq=X//*//../*//Y with w many intermediary "*"s and a matching data path, p, of length $l \geq w + 2$, by just one iteration of Yen's k^{th} shortest loopless paths algorithm, we are able to identify a compact representation of $O(l^w)$ many sub-results.

Note that a particular advantage of this compact, clustered representation is that the sub-results do not need to be de-clustered for the join operation. Let us reconsider the query q=A//*[//B]//C and the corresponding match shown in Figure 18. The same match with the same cost would be obtained when paths from As to Bs and from As to Cs are enumerated independently and put together, based on the data node corresponding to A. It is easy to generalize this example to twig queries with more than one "*" wildcard symbols. In general, we translate twig queries with "*"s to twig queries without any "*". The sub-result combination step (Section 3.4.1) eliminates false hits. A post-processing step after the HR-Join based query plan can then identify the node IDs of the data nodes matching wildcards.



(a) A "*" query (b) Query rewritten without "*"

(c) Match (d) False-hit

Figure 18: (a) A twig query with a "*" wildcard and (b) its no-"*" version. (c) A result to the original query where the nodes E and F match the "*"; this result would also be returned for the rewritten query: the common ancestors of the nodes B and C (except A itself) are matches to "*". (d) A false-hit for the rewritten query; these false hits can be eliminated from the result stream efficiently

5. EXPERIMENTS

In this section, we evaluate the proposed schemes for horizon-based ranked join and twig query evaluation on weighted graph data. The experiments reported in this section ran on a 2GHz Pentium with 1GB main memory.

5.1 HR-Join and M-Way HR-Join

In Section 3.4, we introduced various behavioral metrics, including *sub-result pruning* and *data overhead*, for HR-Join operators. We also discussed the expected behavior patterns for HR-Join and M-Way HR-Join operators. The amount of input pruning depends not only on the workings of the operator, but also on the total number of sub-results that there exist. We leave the evaluation of this to Section 5.2, where we report results for queries executed on graph data. Here we focus on experimental evaluation of the relationship between *time to compute top-k results*, *data overhead*, and *intermediary result pruning* behaviors.

These patterns are visualized in Figure 19. In this figure, HR stands for the plan in Figure 12 and HRM stands for the M-way HR-Join based plan (with a single, shared *horizon*) in Figure 16. The figure shows the behavior of the plans for two significantly different join-selectivity distributions: $\sim$ 10% (where the likelihood of matches between the sub-paths is around 10%) and 1-to-1 (where each sub-result matches at most one sub-result in the other streams).

Figure 19(a) compares the data overhead both for input streams and for the intermediary data. According to this figure, as predicted in Section 3.5, HRM join fails to prune intermediary data which do not contribute to the final result. On the other hand, it ensures that all the input data considered by the join block is necessary. The intermediary data overhead is especially high for the $\sim$ 10% join selectivity, which results in a higher number of matches between input data. In contrast, while the HR setup regulates the intermediary data well, it leaks some unnecessary inputs from the input streams to the join block. The reason for the leakage is provided in Figure 19(b). For the HR-Join setup there is a larger variation of *horizon* values computed by the lower-join operators. The horizon values computed by HRM, on the other hand, are closer to the final results themselves. This means that the likelihood of allowing unnecessary inputs to the join process is less in HRM than in the HR setup. The count of *horizon*s (i.e., amount of punctuations that needs to be managed) is also higher in HR than in HRM.

Consequently, the choice between whether to use HR-join or M-way HR-Join depends on the join behavior. In general, since M-way HR-Join involves less punctuation management, it is desirable over HR-Join based schemes. When the join selectivity leads to high join costs and large numbers of candidate results are to be maintained in the result sieve, HR-Join only plans may prove to be better (Figure 19(c)).

5.2 Ranked Twig Query Processing

In this set of experiments, we used the FICSR weighted graph data[38], which represents the integration of a pair of taxonomies with high degree of misalignments. The weights on the edges represent the degrees of *disagreements* between the sources, as well as the user feedback (Figure 1). Since, despite the mis-alignments, there are more parts of the taxonomies matching than mis-matching, the weight distribution of the edges show a Zipfian-like distribution, with more costs closer to 0 than closer to 1. The queries on the integrated taxonomies involved reachability neighborhoods of $\sim$ 125, 250, and 500 data nodes. Each data point is obtained by averaging the results of 10 random twig queries.

Figure 20(a) shows that, as expected, the cost of Yen's algorithm is linear in the number of paths enumerated. Importantly, the inter-arrival time of paths was less than 100ms.

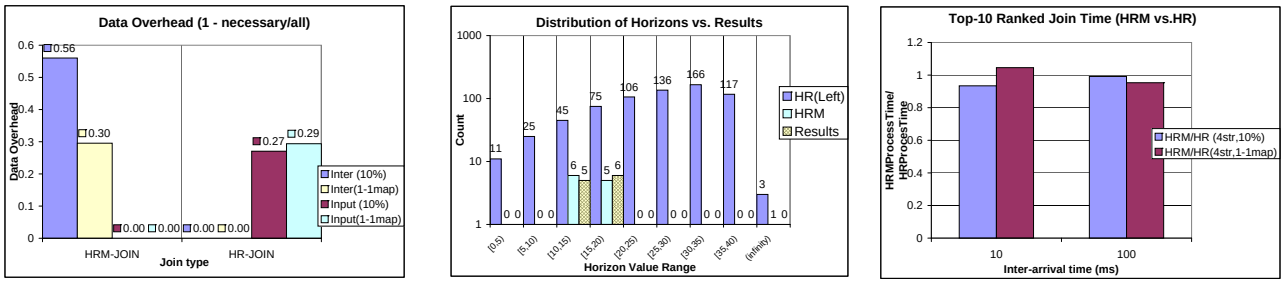
Figure 20(b) plots the time between consecutive outputs of the HR-Join operator (for random queries of type `A[//B]//C`). The neighborhood size affects the speed of the output since the time to identify the input paths increases with the size of the reachability graph (Figure 20(a)). Therefore, the degree of input pruning becomes more important in large neighborhoods. Figure 20(c) plots the degree of pruning obtained by using smaller ks (as opposed to k=20, in this figure). As expected, the degree of pruning is directly correlated with how small k is.

5.3 "*" Wildcards and Horizon Tightening

As discussed in Section 4.4, results to the rewritten queries generally encode more than one match to the original query. Thus, few matches to the rewritten query can be sufficient to obtain all top-10 matches. In Figure 21, we plot results for different tf values for queries of type `A//*[//B]//C`. We also plot results for rewritten queries, of type `A[//B]//C`, for top-10 *distinct results* as well as for top-10 *distinct encodings*.

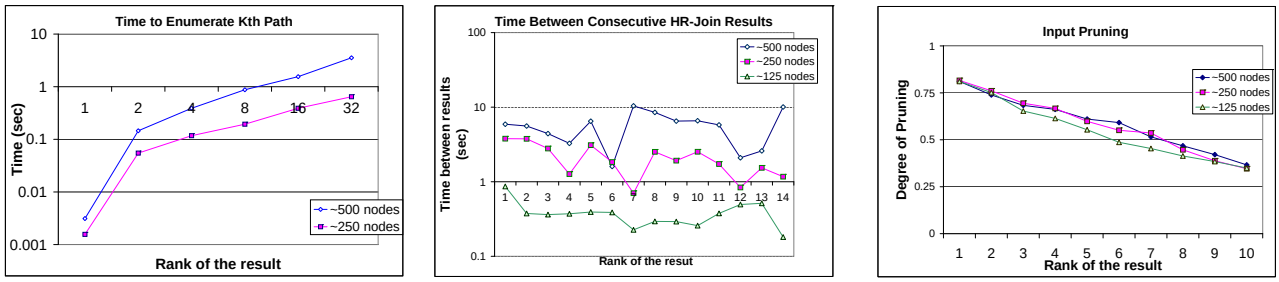
Figure 21(a) shows that smaller *horizon* tightening factors (tfs) provide significant time savings, since they allow early punctuations. Rewritten queries, on the other hand, execute significantly faster than the original query (even for very low *horizon* tightening factors, e.g. 0.5). This is true especially for the first few distinct encodings they return.

Figure 21(b) shows the costs of the enumerated results for the original "*" query, for different tf values. The figure also shows the cost of *distinct results* and *distinct encodings* obtained through rewriting. From this figure, we can see that the costs of the distinct results returned by the rewritten query are significantly better than the others. Original queries with tf values close to 1.0 provide costs similar to those of the *distinct encodings* returned by rewritten queries. Early punctuation (even for tf values close to 1.0) prevents



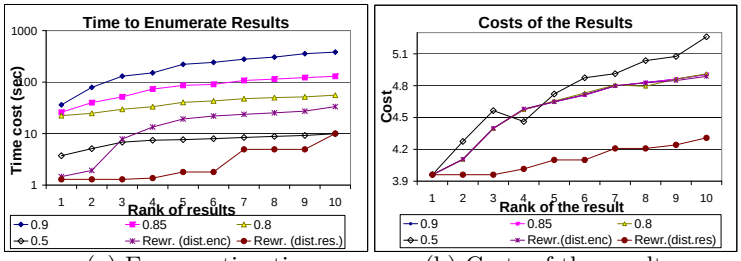
(a) Data overhead (input, intermediary) (b)*Horizon* and result distribution (c) Time to top-10 (HRM/HR)

Figure 19: Comparison of HR-Join and M-Way HR-Join operators



(a) Path enumeration time using [37] (b) Time between consecutive results (c) Pruning (relative to the 20^{th} result)

Figure 20: Twig query processing results on weighted graph data for queries of type `A[//B]//C`



(a) Enumeration time (b) Costs of the results

Figure 21: Effects of tf on queries of type `A//*[//B]//C` on reachability graphs of ∼250 nodes

enumeration of all relevant paths (thus provides time savings, but also eliminates some results). Significantly lower tf values (eg. 0.5) miss further opportunities and negatively affect the result order and costs.

6. RELATED WORK

Query Processing on Trees and Graphs. In the Introduction, we have highlighted that many existing XML query processors [9, 14, 4] rely on the tree-like structure of the XML data and require that the ancestor and descendant lists be available in a structurally sorted order. Structural join schemes also exploit pre-computed or on-the-fly index structures (such as B^+ trees and its variations [9, 15, 29, 10]) to skip unpromising elements relying on the tree-like structure of the data. Path-based index structures on semi-structured data and on rooted-graphs include FB-Index [31], APEX [16], Dataguides [24], T-Index [35], and other covering indices [40]. XML filtering schemes, such as [11] and others, also leverage the tree-like structure of the XML data for efficiency. Such approaches for pruning structurally irrelevant nodes become quickly infeasible when we are not only interested in shortest-paths, but all paths in progressively increasing order of cost.

Keyword Queries on XML/Graph Data. Recently, there has been growing interest in integrating information-retrieval style, keyword-based access into databases. XRank [25] and ObjectRank [6] propose algorithms to compute node rankings for keyword-style database queries. XSEarch [17], a semantic search engine for XML data, relies on extended information retrieval techniques in ranking its results. [42] proposes a new operator for XML query processing, where the result type of the user query is not explicitly specified, but the query returns the most *specific* result in the database. Unlike the above papers, which all aim to identify the *best* node matching a given query, RIU [34], BANKS-I [7] and BANKS-II[30], and DPBF [19] are algorithms trying to find small-size subtrees connecting all the nodes that match any keyword in the query.

Ranked Query Processing. Ranked query processing is important in many application domains, including information retrieval and multimedia [1]. [21] and [13] propose ranked query evaluation algorithms, both of which assume that individual sources can progressively output sorted results and also enable random-access. These algorithms also assume that the query has a monotone combined scoring function. [12] presents *approximate* ranked query processing techniques for cases where not all sub-queries are able to return ordered results. [22] and [32] recognize that there may be cases where random accesses are impossible and present algorithms, under monotonicity assumption, to enumerate top-k objects without random access. They augment *monotonicity* with an *upper bound* principle to enable bounding of the maximum possible score of a partial result.

[20] focuses on the optimization of the *top-k* queries. [27] introduces a *rank-join* operator that can be deployed in existing query execution interfaces. Under the monotonicity assumption, they also propose a score-guided join strategy to minimize the number of inputs to consider. J^* [36] uses an A^* based heuristic and the monotonicity assumption to help guide the navigation in the ranked join space. [43] proposes *ranked join indices* for the efficient evaluation of ranked query result. [2] and [32] extend the relational algebra to support ranking as a first-class construct. [32]

also presents a pipelined and incremental execution model of ranking query plans. In [33], authors extend ranked query processing to aggregate queries. [47] models top-k querying as a k-constrained optimization problem, where a goal function includes both a boolean constraint characterizing the data of interest and a quantifying function which acts as the numeric optimization target.

7. CONCLUSIONS

Weighted, graph-based representation of the data is common in many application domains. Yet, while the problems of (a) keyword-based access to data graphs, (b) structural joins for efficient querying of XML data, and (c) top-k access to data for IR, multimedia, and web applications have been considered independently, ours is the first work which faces and tackles these three challenges with the goal of providing querying and retrieval over weighted data graphs. We showed that, while the such queries can be treated as *ranked structural-joins*, the critical monotonicity property does not hold. We also highlighted that top-k queries cannot benefit from existing path-index structures. To address the first shortcoming, we introduced a *sum-max monotonicity* property and built a self-punctuating *horizon-based ranked join* (HR-Join) operator for twig query evaluation. To address the second challenge, we proposed progressive path-based sub-query evaluation schemes. We also presented a *horizon* tightening approach to deal with the potentially large number of paths in the graph. As verified by the experiments, HR-Join, used along with a progressive shortest simple path enumeration module, enables efficient processing of top-k twig queries on weighted graph data.

8. REFERENCES

[1] S. Adali, P. A. Bonatti, M. L. Sapino, V. S. Subrahmanian. A multi-similarity algebra. In *SIGMOD*, 1998.

[2] S. Adali, C. Bufi, and M. L. Sapino. Ranked relations: Query languages and query processing methods for multimedia. *Multimedia Tools and Applications*, 24(3):197–214, 2004.

[3] S. Agrawal, S. Chaudhuri, G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[4] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[5] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[6] A. Balmin, V. Hristidis, Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, 2004.

[7] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[8] S. Brin, L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[9] N. Bruno, D. Srivastava, N. Koudas. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[10] K. S. Candan, M. E. Dönderler, Y. Qi, J. Ramamoorthy, J. W. Kim. Fmware: Middleware for efficient filtering and matching of xml messages with local data. In *Middleware*, 2006.

[11] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, D. Agrawal. Afilter: Adaptable xml filtering with prefix-caching and suffix-clustering. In *VLDB*, 2006.

[12] K. S. Candan, W.-S. Li, M. L. Priya. Similarity-based ranking and query processing in multimedia databases. *Data & Knowledge Engineering*, 35(3):259–298, 2000.

[13] S. Chaudhuri, L. Gravano, A. Marian. Optimizing top-k selection queries over multimedia repositories. *TKDE*, 16:992–1009, 2004.

[14] T. Chen, J. Lu, T. Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD*, 2005.

[15] S. Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, 2002.

[16] C.-W. Chung, J.-K. Min, K. Shim. Apex: An adaptive path index for xml data. In *SIGMOD*, 2002.

[17] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, 2003.

[18] P. F. Dietz. Maintaining order in a linked list. In *STOC*, 1982.

[19] B. Ding, J. Yu, S. Wang, L. Qing, X. Zhang, X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[20] D. Donjerkovic, R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, pages 411–422, 1999.

[21] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[22] R. Fagin, A. Lotem, M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[23] N. Fuhr, K. Grosjohann. XIRQL: A query language for information retrieval in XML documents. In *Research and Development in Information Retrieval*,2001.

[24] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[25] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. *SIGMOD*, 2003.

[26] V. Hristidis, Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[27] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB03*, 2003.

[28] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *The VLDB Journal*, 11(4):274–291, 2002.

[29] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *ICDE*, 2003.

[30] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[31] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[32] C. Li, K. C. Chang, I. F. Ilyas, S. Song. Ranksql: Query algebra and optimization for relational topk queries. In *SIGMOD*, 2006.

[33] C. Li, K. C.-C. Chang, I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *SIGMOD*, 2006.

[34] W.-S. Li, K. S. Candan, Q. Vu, D. Agrawal. Retrieving and organizing web pages by information unit. In *WWW*, 2001.

[35] T. Milo, D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

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

[37] M. Pascoal, E. Martins. A new implementation of yen's ranking loopless paths algorithm. *4OR – Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2):121–134, 2003.

[38] Y. Qi, K. S. Candan, M. L. Sapino. Ficsr: Feedback-based InConSistency Resolution and query processing on misaligned data sources. In *SIGMOD*, 2007.

[39] Y. Qi, K. S. Candan, M. L. Sapino, K. Kintigh. Using QUEST for integrating taxonomies in the presence of misalignments and conflicts. In *SIGMOD Demos*, 2007.

[40] P. Ramanan. Covering indexes for xml queries: Bisimulation - simulation = negation. In *VLDB*, 2003.

[41] G. Reich, P. Widmayer. Approximate minimum spanning trees for vertex classes. Technical report, Inst. fur Informatik, Freiburg Univ., 1991.

[42] A. Schmidt, M. L. Kersten, M. Windhouwer. Querying XML documents made easy: Nearest concept queries. In *ICDE*, 2001.

[43] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, D. Srivastava. Ranked join indices. In *ICDE*, 2003.

[44] Xpath. http://www.w3.org/TR/xpath, 1999.

[45] Xquery. http://www.w3.org/TR/xquery, 2006.

[46] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

[47] Z. Zhang, S. Hwang, K. C. Chang, M. Wang, C. A. Lang, Y. Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *SIGMOD*, 2006.