Efficiently Evaluating Skyline Queries on RDF Databases

Ling Chen, Sidan Gao, and Kemafor Anyanwu

Semantic Computing Research Lab, Department of Computer Science
North Carolina State University
{lchen10,sgao,kogan}@ncsu.edu

Abstract. Skyline queries are a class of preference queries that compute the pareto-optimal tuples from a set of tuples and are valuable for multi-criteria decision making scenarios. While this problem has received significant attention in the context of single relational table, skyline queries over joins of multiple tables that are typical of storage models for RDF data has received much less attention. A naïve approach such as a *join-first-skyline-later* strategy splits the join and skyline computation phases which limit opportunities for optimization. Other existing techniques for multi-relational skyline queries assume storage and indexing techniques that are not typically used with RDF which would require a preprocessing step for data transformation. In this paper, we present an approach for optimizing skyline queries over RDF data stored using a vertically partitioned schema model. It is based on the concept of a "*Header Point*" which maintains a concise summary of the already visited regions of the data space. This summary allows some fraction of non-skyline tuples to be pruned from advancing to the skyline processing phase, thus reducing the overall cost of expensive dominance checks required in the skyline phase. We further present more aggressive pruning rules that result in the computation of *near-complete* skylines in significantly less time than the complete algorithm. A comprehensive performance evaluation of different algorithms is presented using datasets with different types of data distributions generated by a benchmark data generator.

Keywords: Skyline Queries, RDF Databases, Join.

1 Introduction

The amount of RDF data available on the Web is growing more rapidly with broadening adoption of Semantic Web tenets in industry, government and research communities. With datasets increasing in diversity and size, there have been more and more research efforts spent on supporting complex decision making over such data. An important class of querying paradigm for this purpose is preference queries, and in particular, skyline queries. Skyline queries are valuable for supporting multi-criteria decision making and have been extensively investigated in the context of relational databases [1][2][3][4][5][6][11][12] but in a very limited way for Semantic Web [8]. A *skyline query* over a data set S with D-dimension aims to return the subset of S which contains the points in S that are not *dominated* by any other data point. For two D-dimensional data points $p(u_1, u_2, u_3, \dots, u_d)$ and $q(v_1, v_2, v_3, \dots, v_d)$, point p is

G. Antoniou et al. (Eds.): ESWC 2011, Part II, LNCS 6644, pp. 123–138, 2011.

said to dominate point q if $p.u_i \succcurlyeq q.v_i$ for all $i \in [1,d]$ and in at least one dimension $p.u_j \succ q.v_j$ where $j \in [1,d]$, $\succcurlyeq$ denotes *better than* or *equal with*, $\succ$ denotes *better than*. For example, assume that a company wants to plan a sales promotion targeting the likeliest buyers (customers that are young with a low amount of debt). Consider three customers *A* (*age 20, debt $150*), *B* (*age 28, debt $200*) and *C* (*age 25, debt $100*). Customer *A* is clearly a better target than *B* because *A* is younger and has less debt. Therefore, we say that *A dominates B*. However, *A* does not *dominate C* since *A* is younger than *C* but has more debt than *C*. Therefore, the skyline result over the customer set {*A, B, C*} is {*A, C*}.

The dominant cost in computing the skyline of a set of *n D*-dimension tuples lies in the number of comparisons that needs to be made to decide if a tuple is or isn't part of the skyline result. The reason is that for a tuple to be selected as being in the skyline, it would need to be compared against all other tuples to ensure that no other tuples dominate it. Further, each tuple pair comparison involves *D* comparisons comparing their values in all *D* dimensions. Consequently, many of existing techniques [1][2][3][4][5][6][11][12] for computing skylines on relational databases focus on reducing the number of tuple pair comparisons. This is achieved using indexing [4][5][6], or partitioning data into subsets [11] where some subsets would contain points that can quickly be determined to be in or pruned from the skyline result.

It is also possible to have skyline queries involving attributes across multiple relations that need to be joined, i.e. multi-relational skyline. This would be a very natural scenario in the case of RDF data since such data is often stored as vertically partitioning relations [7]. However, there are much fewer efforts [11][12][16] directed at evaluating skylines over multiple relations. A common strategy, which was also proposed in the context of preference queries on the Semantic Web [8], is to first join all necessary tables in a single table and then use a single table skyline algorithm to compute the skyline result, i.e. *join-first-skyline-later* (JFSL). A limitation of the JFSL approach is that the join phase only focuses on identifying joined tuples on which skyline computation can then be done. It does not exploit information about the joined tuples to identify tuples that are clearly not in the skyline result. Identifying such tuples would allow pruning them from advancing to the skyline phase and avoid the expensive dominance checks needed for skyline computation on those tuples. Alternative techniques to the JFSL approach [4][5][6][11][12][16] employ specialized indexing and storage schemes which are not typical in RDF data and require preprocessing or storage in multiple formats.

1.1 Contributions

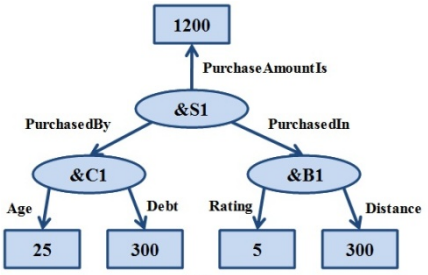
This paper proposes an approach for efficient processing of skyline queries over RDF data that is stored as vertically partitioned relations. Specifically, we propose

- The concept of a Header Point that maintains a concise summary of the already visited region of the data space for pruning incoming non-skyline tuples during join phase. This improves efficiency by reducing number of comparisons needed during later skyline processing phase.

- A complete algorithm and two near-complete algorithms based on an approach that interleaves the join processing with some skyline computation to minimize the number of tuples advancing into skyline computation phase. The near-complete algorithms compute about 82% of skyline results in about 20% of the time needed for the complete algorithm.
- A comprehensive performance evaluation of the algorithms using datasets with different types of data distributions generated by a benchmark data generator.

2 Background and Problem Statement

Assume that we have a company's data model represented in RDF containing statements about *Customers, Branches, Sales and relationships* such as *Age, Debt, PurchasedBy etc.* Figure 1 (a) shows a sample of such database using a graph representation.




```
SELECT *
WHERE { ?Sale       PurchasedBy   ?Customer
        ?Customer   Age           ?ageVal
        ?Customer   Debt          ?debtVal
        ?Sale       PurchasedIn   ?Branch
        ?Branch     Distance      ?distVal }
SKYLINE OF ?ageVal MIN, ?debtVal MIN,
           ?distVal MIN
```

(b)

Fig. 1. Example RDF Graph Model and Skyline Queries

Consider again the example of sales promotion targeting the young customers with less debt. Also assume that company would like to focus their campaigns on customers that live close to some branch. We express such a query using an extended SPARQL as shown in Figure 1 (b). The properties in front of MIN/MAX keywords are the skyline properties (dimensions) to be considered during the skyline computation. The MIN/MAX keyword specifies that we want the value in the corresponding property to be minimized/maximized. We now formalize the concept of a *skyline graph pattern* that builds on the formalization of SPARQL graph pattern queries.

An RDF *triple* is 3-tuple (s, p, o) where *s* is the *subject*, *p* is the *predicate* and *o* is the *object*. Let *I*, *L* and *B* be the pairwise disjoint infinite set of *IRIs*, *Blank nodes* and *Literals*. Also assume the existence of an infinite set *V* of variables disjoint from the above sets. A *triple pattern* is a query pattern such that *V* can appear in *subject*, *predicate* or *object*. A *graph pattern* is a combination of triple patterns by the binary operators *UNION*, *AND* and *OPT*. Given an RDF database graph, a solution to a graph pattern is a set of substitutions of the variables in the graph pattern that yields a subgraph of the database graph and the solution to the graph pattern is the set of all possible solutions.

Definition 1 (Skyline Graph Pattern). *A Skyline Graph Pattern is defined as a tuple (GP, SK) where GP is a graph pattern and SK is a set of skyline conditions* $\{sk_1, sk_2, \ldots, sk_m\}$. sk_i *is of the form* $fn_i(p_i)$ *where* fn_i *is either min() or max() function and* p_i *is a property in one of the literal triple patterns in GP. The Solution to (GP, SK) is an ordered subset* $RS = [S_i, S_j, \ldots, S_k] \subseteq S$ *where* S_i *denotes the solution to a basic graph pattern (an RDF graph with variables) and the following conditions hold: (i) each solution* $S_p \in RS$ *is not dominated by any solution in* S; *(ii) each solution* $S_q \in \{S - RS\}$ *is dominated by some solution in* S.

3 Evaluating the Skyline over the Join of Multiple-Relations

Vertically partitioned tables (VPT) are a common storage model for RDF data. A straightforward approach to compute the skyline over a set of vertically partitioned tables $VPT_1, VPT_2, \ldots, VPT_d$ is as follows: (i) join $VPT_1, VPT_2, \ldots, VPT_d$ into a complete single table *T* (the determination of dominance between two tuples cannot be made by looking at only a subset of the skyline properties); (ii) compute skyline over this single table *T* by using any single-table skyline algorithm, such as BNL (Block-Nested-Loop). We call this approach as "*Naive*" algorithm. "*Naive*" algorithm maintains only a subset of all already joined tuples (*candidate list*) against which each newly joined tuple is compared to determine its candidacy in the skyline result. However, this approach does not fully exploit the information about the joined tuples and requires too many comparisons to determine one tuple's candidacy in skyline result. Our approach based on the concept of a *Header Point* improves upon this by using information about already joined tuples to make determinations about (i) whether a newly joined tuple could possibly be a member of the skyline and (ii) whether there is a possibility of additional skyline tuples to be encountered in the future. The former allows for pruning a tuple from advancing to the skyline (SL) phase where it would incur additional cost of comparisons with several tuples in a skyline candidate list. The latter allows for early termination.

3.1 Header Point and Its Prunability

Our approach is based on splitting the join phase into iterations where information about earlier iterations is summarized and used to check skyline candidacy of tuples joined in later iterations. In each *join iteration*, we need to join each 2-tuple in each VPT to their corresponding matching 2-tuples in all of the other VPTs. Let J_i be the *table pointer* pointing to the subject value (sub_j) of the *j*th triple tr_j in VPT_i and a *join iteration* would be:

$$\bigcup_{\text{i=1 to d}} \text{join of } tr_j \text{ to matching tuples in } VPT_k (\, k \neq i)$$

In other words, at the end of a join iteration we would have computed *d* tuples and each tuple is based on the 2-tuple pointed to by table pointer in some dimension VPT.

A *Header Point* summarizes the region of data explored in earlier join iterations. It enables a newly joined tuple in the subsequent join iteration to be compared against this summary rather than multiple comparisons against the tuples in the skyline candidate list.

Definition 2 (Header Point). *Let* $\{t_1, t_2, \ldots, t_d\}$ *be the set of tuples in the jth join iteration. A Header Point of the computation is a tuple of* $< f_{n_j}(\{t_i[1]\}), f_{n_j}(\{t_i[2]\}), \ldots, f_{n_j}(\{t_i[d]\}) >$ *where* f_{n_j} *is either min() or max() function. We call the tuples that form the basis of the Header Point (i.e. the* t_i*s), Header Tuples.*

To illustrate the advantage of the header point concept, we will use a smaller version of our motivating example considering only a graph sub pattern with skyline properties, *Age* and *Debt*. We will assume that data is organized as VPT and indexed by Subject (SO) and by Object (OS). Using the OS index, the triples can be accessed in decreasing order of "goodness" when minimizing/maximizing skyline properties, i.e. in increasing/decreasing order of object values. Let us consider the earliest join iteration involving the first tuples of each relation. Figure 2 (a) shows the table pointers (J_{Age} and J_{Debt}) for the two relations and the two red lines show the matching tuples to be joined resulting in the tuples *T1* (*C1, 25, 2800*) and *T2* (*C13, 32, 800*) shown in Figure 2 (b). Since these tuples are formed from the best tuples in each dimension, they have the best values in their respective dimensions, therefore no other tuples dominate them and they are part of the skyline result. We can create a header point based on the worst values (here, the largest values) in each dimension (*Age* and *Debt*) across all the currently joined tuples resulting in a tuple *H* (*32, 2800*). *T1* and *T2* are called Header Tuples.

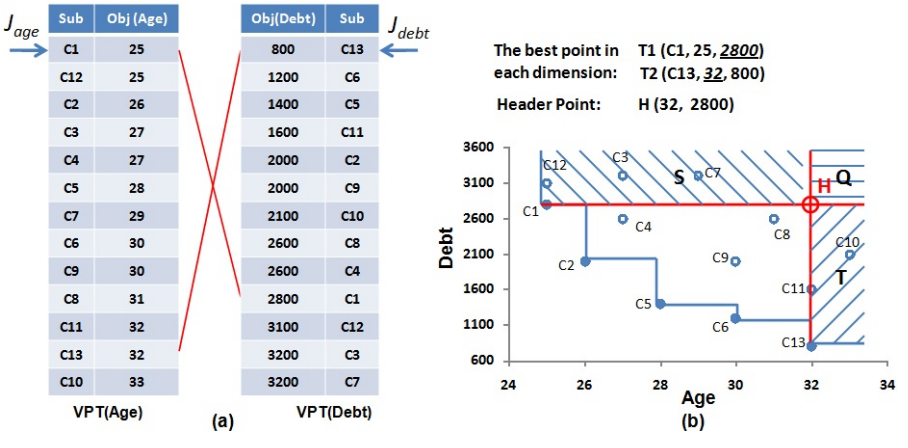


Fig. 2. Formation of Initial Header Point and its Prunability

Our goal with the header point is to use it to determine whether newly joined tuples in subsequent iterations should be considered as candidate skyline tuples or be pruned from advancing to the expensive skyline processing phase. For example, assume that in the next join iteration we compute the next set of joined tuples by advancing the table pointer to the next tuple in each VPT. Then, one of the resulting joined tuples is (*C12, 25, 3100*). For the rest of the paper, we will use *C12* to denote the tuple (*C12, 25, 3100*). However, the current candidate skyline list { (*C1, 25, 2800*), (*C13, 32, 800*) } contains a tuple (*C1, 25, 2800*) that clearly dominates *C12*, therefore *C12* should not be considered as a candidate skyline tuple. Further, we should be able to

use the information from our header point to make this determination rather than compare every newly joined tuple with tuples in the candidate skyline list. We observe the following relationship between newly joined tuples and the header point that can be exploited: *If a newly joined tuple is worse than the header point in at least one dimension, then that new tuple cannot be a skyline tuple and can be pruned*. Applying this rule to our previous example, we will find that the tuple *C12* is worse than our header point *H* in the *Debt* dimension. Therefore, it can be pruned. In contrast, the other joined tuple (*C6, 30, 1200*) is not worse than *H* in at least one dimension (in fact, it is better in two dimensions) and is not pruned. Figure 2 (b) shows the relationships between these points in a 2-dimensional plane: any subsequent joined tuple located in the regions of *S*, *Q* and *T* can be pruned by header point *H*. We now make a more precise statement about the relationship between a header point and tuples created after the creation of the header point.

Lemma 1. *Given a D-dimensional header point* $H = < h_1, h_2, \ldots, h_D >$, *any "subsequent " (i.e. a point constructed after the current header point) D-tuple whose values are worse than H in at least D - 1 dimensions, are* ***not*** *skyline points.*

Proof. Let $P = (p_1, p_2, \ldots, p_D)$ be a new tuple that is worse than the current header point $H = (h_1, h_2, \ldots, h_D)$ in at least $D - 1$ dimensions but is a candidate skyline point. Let H be the header point that was just formed during the construction of the most recently computed set $B = \{ B_1 = (nbv_1, q_2, \ldots, q_D)$, $B_2 = (t_1, nbv_2, \ldots, t_D)$, $\ldots$, $B_d = (s_1, s_2, \ldots, nbv_D)$ $\}$ of candidate skyline points (header tuples), where nbv_i denotes the next best value in VPT_i. Recall that B consists of the last D tuples that resulted from the join between best tuples in each dimension and a matching tuple in each of the other dimensions.

Assume that the dimensions in which *P* has worse values than the header point *H* are dimensions 2 to *D*. Then, *H* "partially dominates" *P* in dimensions 2 to *D*. Further, since the header point is formed from the combination of the worst values in each dimension across the current header tuples, *P* is also partially dominated by the current header tuples which are also current candidate skyline tuples. Therefore, the only way for *P* to remain in the skyline is that no candidate skyline tuples dominate it in the only remaining dimension, dimension 1. However, the header point tuple $B_1 = (nbv_1, q_2, \ldots, q_d)$ which is currently a candidate skyline tuple has a dimension-1 value - nbv_1 that is better than p_1. This is because the values are sorted and visited in decreasing order of "goodness" and the tuple $B_1$was constructed before *P*, so the value nbv_1 must be better than p_1. This means that B_1 is better than *P* in all dimensions and therefore dominates *P*. Therefore, *P* cannot be a skyline tuple which contradicts our assumption.

3.2 RDFSkyJoinWithFullHeader (RSJFH)

We now propose an algorithm *RDFSkyJoinWithFullHeader (RSJFH)* for computing skylines using the *Header Point* concept. In the *RSJFH* algorithm, a join iteration proceeds as follows: *create a joined tuple based on the tuple pointed by the table pointer for dimension i. If a resulting joined tuple is pruned, then advancing table i's pointer until it points to a tuple whose joined tuple is not pruned by the current header point. This process is repeated for each dimension to create a set of d header*

tuples. Since the header point is formed using the worst values in each dimension among the joined tuples, it may represent very loose boundary conditions which will significantly reduce its pruning power. This occurs when these worst values are among the worst overall in the relations. In our example, this occurs in the first join iteration with the construction of the initial header point *H*(*32, 2800*). To strengthen the pruning power of the header point, we can update its values as we encounter new tuples with better values i.e. the next set of *D* tuples whose set of worse values are better than the worse values in the current header point. These tuples will become the next iteration's header tuples.

Figure 3 (a) shows the second join iteration where new header tuples are used to update the Header Point. The next tuple in *Age* VPT is (*C12, 25*) and its joined tuple is (*C12, 25, 3100*). Compared with *H*, *C12* can be pruned since it is worse than *H* in at least *D-1* dimensions (*D* is 2). *RSJFH* advances the table pointer to the next tuple in the *Age* table, (*C2, 26*) whose joined tuple is (*C2, 26, 2000*). Compared with *H*, *C2* is not pruned and this tuple is adopted as a header tuple. Then, *RSJFH* moves to the next VPT *Debt* where the next tuple is (*C6, 1200*) and its joined tuple is (*C6, 30, 1200*). Compared with *H*, *C6* is not pruned. Now, there is one header tuple from each VPT and the header point can be updated to *H'* (*30, 2000*). Similarly, in the third join iteration (Figure 3 (b)), *RSJFH* checks the subsequent joined tuples in tables *Age* and *Debt* and finds (*C5, 28, 1400*) and (*C5, 28, 1400*) are the next header tuples in tables *Age* and *Debt* respectively. Then, the header point is updated to *H''* (*28, 1400*) based on these two header tuples.

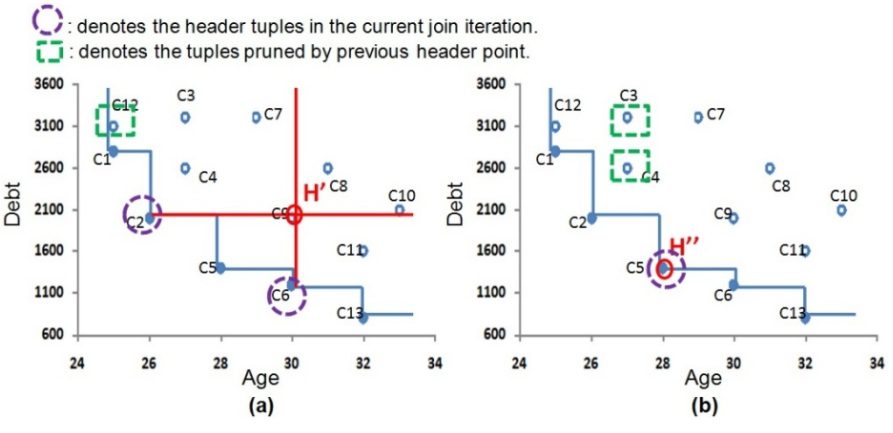


Fig. 3. Updating Header Points

3.2.1 Termination Phase and Post Processing

The *RSJFH* terminates the search for additional candidate skyline tuples when either (i) the header point is not updated or (ii) either one of the table pointers J_i advances down to the end of VPT_i.

Lemma 2. *Let* $H =< h_1, h_2, \ldots, h_D >$ *be the last D-dimensional header point computed during the last join iteration i.e. the joining process that resulted in the computation of the last D candidate skyline tuples. If during the current join iteration*

the header point is not updated or either one of the table pointers J_i advances down to the end of VPT_i, then the search for additional candidate skyline tuples can be terminated losslessly.

Proof. Recall that during a join iteration, we pick the next best tuples that are not pruned in each dimension and join with other dimensions to form the next D candidate skyline tuples. So each resulting tuple should contain the next best value in some dimension, i.e. $B = \{B_1 = (nbv_1, q_2, \dots, q_D), B_2 = (t_1, nbv_2, \dots, t_D), \dots, B_d = (s_1, s_2, \dots, nbv_D)\}$. If after computing the set B, the header point is not updated, it implies that each nbv_i is worse than the corresponding h_i in H. It is clear that all these tuples can be pruned because their best dimension values are worse than our current worst seen values in our header tuple, resulting in that the other dimensions are clearly also worse. Thus, the header tuple dominates all of them. Further, since the tuples in each dimension are ordered in the decreasing order of "goodness", the next set of best values cannot be better than those currently considered. Therefore, no better tuples can be found with further scanning of the tables. When either one of the table pointers J_i advances down to the end of VPT_i then all the values for that dimension have been explored. Therefore, no new D-tuples can be formed and the search for additional candidate skyline tuples can be terminated losslessly.

Algorithm 1. RDFSkyJoinWithFullHeader (RSJFH)

INPUT: *n* ***VPTs*** which are sorted according to object value, ***VPTList***.
OUTPUT: A set of skyline points.
1.**Initialization** // to get first header point
2. read first tuple in each *VPT* and hash join to get *n* complete tuple *Ti*.
3. take the worst value in each dimension among *Ti* (i=1, 2,.., n) to compute Header Point *H*
4.**While** *H* is updated **or** pointers J_t is not pointing to the end of VPT_t **do**
5. **for each** VPT $t \in$ *VPTList*
6. read one tuple and hash join to get complete tuple *T* and compare *T* with *H*
7. **if** *T* is prunable by *H*
8. *T* is pruned
9. **else**
10. *T* is a Header Tuple for updating *H* and is inserted into Candidate List *C*
11. **end for**
12. update Header Point *H* by Header Tuples
13**. end while**
14.BNLCalculate(*C*). // use BNL skyline algorithm to compute skyline results

Discussion. Intuitively, the header point summarizes neighborhoods that have been explored and guides the pruning of the tuples in neighborhood around it during each iteration. However, since *RSJFH* uses the worse points in each dimension, and prunes tuples that are really worse (worse in *d-1* dimensions) than the header point, it only allows *conservative* pruning decisions. Therefore, some non-skyline points can still be inserted into the candidate skyline list. For example, assume that *RSJFH* has just processed the following tuples into the candidate skyline list {(*25, 4000*), (*28, 3500*), (*30, 3000*)} and computed the header point (*30, 4000*). Then, an incoming tuple (*29, 3750*) would be advanced to the next stage because it is better than the header point in

both dimensions. However, it is unnecessary to advance the tuple (*29, 3750*) into the next stage of the computation because it would eventually be dropped from candidate skyline list since the tuple (*28, 3500*) dominates this tuple.

4 Near-Complete Algorithms

We can try to improve the header point to allow for more aggressive pruning. We may however risk pruning out correct skyline tuples. In the following section, we propose a strategy that strikes a balance between two objectives: increasing the aggressiveness of pruning power of the header point and minimizing the risk of pruning correct skyline tuples. We posit that in the context of the Web, partial results can be tolerated particularly if the result set is "near-complete" and can be generated much faster than the "complete" set. The approach we take is based on performing *partial* updates to the header point after each join iteration rather than updating all the dimensions.

4.1 RDFSkyJoinWithPartialHeader (RSJPH)

Definition 3 (Partial Header Point Update). *Let H be the header point generated in the previous join iteration and* $t_1, t_2, \ldots, t_n$ *be the tuples in the current join iteration. A partial update to the header point means that for the ith dimension of header point, the value is updated only if the worst value of* $t_1, t_2, \ldots, t_n$ *in the ith dimension is better than the ith dimensional value in H.*

This implies that if all values in the *i*th dimension are better than the *i*th dimensional value of the header point, then the *i*th dimension of the header point is updated with worst value as before; otherwise, the *i*th dimension is not updated. Thus, the header point is aggressively updated by the improving (or advancing) dimension values of the joined tuples in the current join iteration.

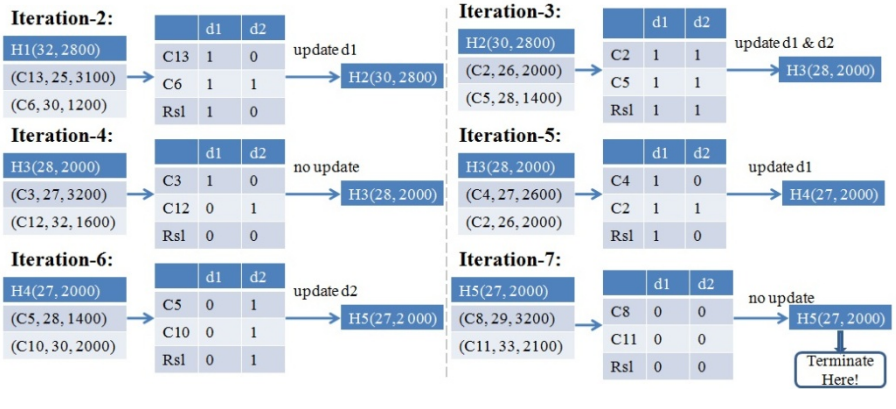


Fig. 4. Partially Update Header Points by Bitmap Computation and its Termination

We now propose an algorithm *RDFSkyJoinWithPartialHeader (RSJPH)* based on the partial update. To efficiently perform a "partial update of header point", *RSJPH* uses *bitmap* representation for *bitwise* operations. Consider the skyline graph sub pattern of *Customer* again. Figure 4 shows the partial update method applied after the first join iteration. In the second iteration, there are two newly joined tuples (*C13, 25, 3100*) and (*C6, 30, 1200*) and a header point *H1*(*32, 2800*) generated in the previous iteration. Since both the dimension-1 values of *C13* and *C6* are better than that of *H1* and only the dimension-2 value of *C6* is better than that of *H1*, header point is partially updated by the worse dimension-1 values of *C13* and *C6*, resulting in *H2*(*30, 2800*). This partial update method is implemented using *Bitmap* computation (bit "1" denotes the dimension that has better value, bit "0" denotes a contrary case and a resulting bit array indicates that in which dimension(s) the header point needs to be updated). Iterations 3 to 6 perform the same way to partially update the header points.

Termination. *RSJPH* revises the termination condition based on the partial update (Line 2 in Algorithm 2). *RSJPH* terminates when there is no update for header point and all the dimensional values in the newly joined tuples are worse than that of current header point. Iteration 7 in Figure 4 shows how *RSJPH* terminates.

Algorithm 2. RDFSkyJoinWithPartialHeader(RSJPH)

INPUT: *n* ***VPT*** which are sorted according to object value, ***VPTList***.
OUTPUT: A set of skyline points.
Define Variables: in n newly-joined tuples $t_1, t_2, \ldots, t_n$, let dimension-*1* value of t_1 be V_1, dimension-*2* value of t_2 be V_2, …, dimension-*n* value of t_n be V_n.

```
1.Initialization // to get first header point H
2.While (H is updated) && (<V1, V2, ..., Vn> is better than H) do
3.      for each VPT t ∈ VPTListI do
4.              read one tuple and hash join to get complete tuple T and compare T with H
5.              use the bitmap representation to record T's better value and worse value
6.              if T is prunable by H then
7.                      T is pruned
8.              else
9.                      T is inserted into Candidate List C
10.     end for
11.     read the bitmap as bit-slice and calculate bit-slice value by bitwise AND operation
12.     for each bit-slice do // partially update H
13.             if bit-slice value is 1 then
14.                     update the corresponding header point value
15.             else
16.                     no update
17.     end for
18.end while
19.BNLCalculate(C).
```

Discussion. In *RSJFH*, we always update the dimensions using the worse values from the header tuples in that iteration regardless of whether those values are better or worse than the current header point values. Essentially, a header point summarizes only the iterations preceding it. In *RSJPH*, a header point may hold "good" pruning values from earlier iterations and only update those dimensions whose values are

improving or advancing. This in a sense broadens the neighborhood that it represents and also allows certain regions in the space to be considered for longer periods than the previous approach. Consequently, we have a header point with more aggressive pruning power. For example, assume that after our previous header point (*30, 4000*), the next iteration would have worse values as (*25, 4500*). However, given our partial update technique, only the first dimension would be updated to 25, resulting in the header point (*25, 4000*). This header point would prune more tuples than the regular header point (*25, 4500*). However, some of the pruned tuples that fall into the gap between (*25, 4500*) and (*25, 4000*) may be skyline tuples, such as (*25, 4250*).

Proposition 1. *Let* $H_1^f, H_2^f, \ldots, H_m^f$ *be the header points generated in RSJFH and A be the set of pruned tuples by all the header points. Similarly, let* $H_1^p, H_2^p, \ldots, H_n^p$ *be the header points generated in RSJPH and B be the set of pruned tuples by all the header points.* H_i^p *is stronger than* H_i^f *, which means* H_i^p *may wrongly prune some skyline points* WP_i *falling into the interval* $<H_i^f, H_i^p>$*. Then,* $A \subset B$ *and the difference set* $B - A = \bigcup_{i=1}^{n} WP_i$.

To avoid this we can relax the pruning check. Rather than generalizing the checking based on number of dimensions, we do checking based on which dimensions were updated and which were not updated.

4.2 Relaxing Prunability of Partially Updated Header Point

Given a header point, if the *i*th dimension is updated in the last iteration, we regard it as an "*updated dimension*"; otherwise, we regard it as "*non-updated dimension*". Given a tuple *p*, if *p* has *n* dimensions whose values are better than that of the header point *h*, we say that *p* has *n better dimensions*. From **Lemma 1**, we can infer that a tuple *p* needs to have at least two *better dimensions* to survive the pruning check. Assume that we have: (1) $H_2(d_1', d_2')$ is a header point partially updated from a full updated header point $H_1(d_1, d_2)$, where $d_1' \succ d_1$ and $d_2' = d_2$, where $\succ$ denotes *better*; Thus, d_1' is the "updated" dimension of H_2 and d_2' is the "non-updated" dimension of H_2; (2) a tuple $p(p_1, p_2)$, where $d_1' \succ p_1$, $p_1 \succ d_1$ and $p_2 \succ d_2'$. Compared to H_2, *p* can be pruned because *p* has only one *better dimension*. However, when compared to H_1, *p* will survive the pruning check since *p* has two *better dimensions* ($p_1 \succ d_1$ and $p_2 \succ d_2$). Since the partial update approach makes the "updated" dimensions of H_2 too good, the tuples that may survive given the fully updated header point H_1, such as *p*, are mistakenly pruned. To alleviate this situation, we relax the pruning condition with the following crosscheck constraint.

Crosscheck. *If an incoming tuple has some dimensional values better than "non-updated" dimension and some dimensional values worse than "updated" dimension, we add this tuple into candidate list. To implement this algorithm, we basically add this additional condition check between Lines 6 and 7 in Algorithm 2. The resulting algorithm is called RDFSkyJoinWithPartialHeader+ (RSJPH+).*

Proposition 2. *Let* H_i^p *be a header point in RSJPH with the "updated" dimension* d_g *that has been updated in iteration i-1 and the "non-updated" dimension* d_b *that has not been updated in iteration i-1. Let p be a new tuple that has failed the pruning*

check by H_i^p. *If p survives the crosscheck condition, i.e.,* $H_i^p.d_g \succcurlyeq p.d_g$ *but* $p.d_b \succcurlyeq H_i^p.d_b$, *p is saved and added into candidate list. We regard the saved tuples by crosscheck in iteration i as the set* CC_i. $CC_i \subseteq WP_i$. *Assume C denotes the set of pruned tuples in RSJPH+. Then,* $C = B - \bigcup_{i=1}^{n} CC_i$ *and* $A \subset C \subseteq B$.

5 Experimental Evaluation

Experimental Setup and Datasets. In this section, we present an experimental evaluation of the three algorithms presented in above sections in terms of scalability, dimensionality, average completeness coverage and prunability. We use the synthetic datasets with independent, correlated and anti-correlated data distributions generated by a benchmark data generator [1]. Independent data points follow the uniform distribution. Correlated data points are not only good in one dimension but also good in other dimensions. Anti-correlated data points are good in one dimension but bad in one or all of the other dimensions. All the data generated by the data generator is converted into RDF format using JENA API and is stored as VPT using BerkeleyDB. All the algorithms are implemented in Java and the experiments are executed on a Linux machine of 2.6.18 kernel with 2.33GHz Intel Xeon and 16GB memory. The detailed experimental results can be found at sites.google.com/site/chenlingshome.

Scalability. Figure 5 (A), (B) and (C) show the scalability evaluation of *RSJFH*, *RSJPH*, *RSJPH+* and *Naïve* for independent, correlated and anti-correlated datasets (1 million to 4 million triples). In all data distributions, *RSJPH* and *RSJPH+* are superior to *RSJFH* and *Naïve*. The difference in execution time between *RSJPH*, *RSJPH+* and *RSJFH* comes from the fact that partial update method makes the header point stronger (i.e. the header point has better value in each dimension and could dominate more non-skyline tuples resulting in stronger prunability) earlier, which terminates the algorithm earlier. For independent data (Figure 5 (A)), *RSJPH* and *RSJPH+* use only about 20% of the execution time needed in *RSJFH* and *Naïve*. The execution time of *RSJFH* increases quickly in the independent dataset with size 4M of triples. The reason for this increase is that the conservativeness of the full header point update technique leads to limited effectiveness in prunability. This results in an increased size for the candidate skyline set and consequently total number of comparisons with Header Point. *RSJPH+* relaxes the check condition in *RSJPH* and so allows more tuples to be inserted into the candidate list explaining the slight increase in the execution time in Figure 5(A). Figure 5 (B) shows that *RSJFH*, *RSJPH* and *RSJPH+* perform better in correlated datasets than in independent datasets. In the correlated data distribution, the header points tend to become stronger earlier than in the case of independent datasets especially when the data is accessed in the decreasing order of "goodness". The reason for this is that the early join iterations produce tuples that are made of the best values in each dimension. Stronger header points make the algorithm terminate earlier and reduce the number of tuples joined and checked against the header point and the size of candidate skyline set. Figure 5 (C) shows particularly bad performance for the anti-correlated datasets which often have the best value in one dimension but the worst value in one or all of the other dimensions. This leads to very weak header points because header points are constructed from worst values of joined

tuples. *RSJFH* have to explore almost the entire search space, resulting in the poor performance shown in Figure 5 (C). Although *RSJPH* seems to outperform the other algorithms, this advantage is attributed to the fact that it computes only 32% of complete skyline result set.

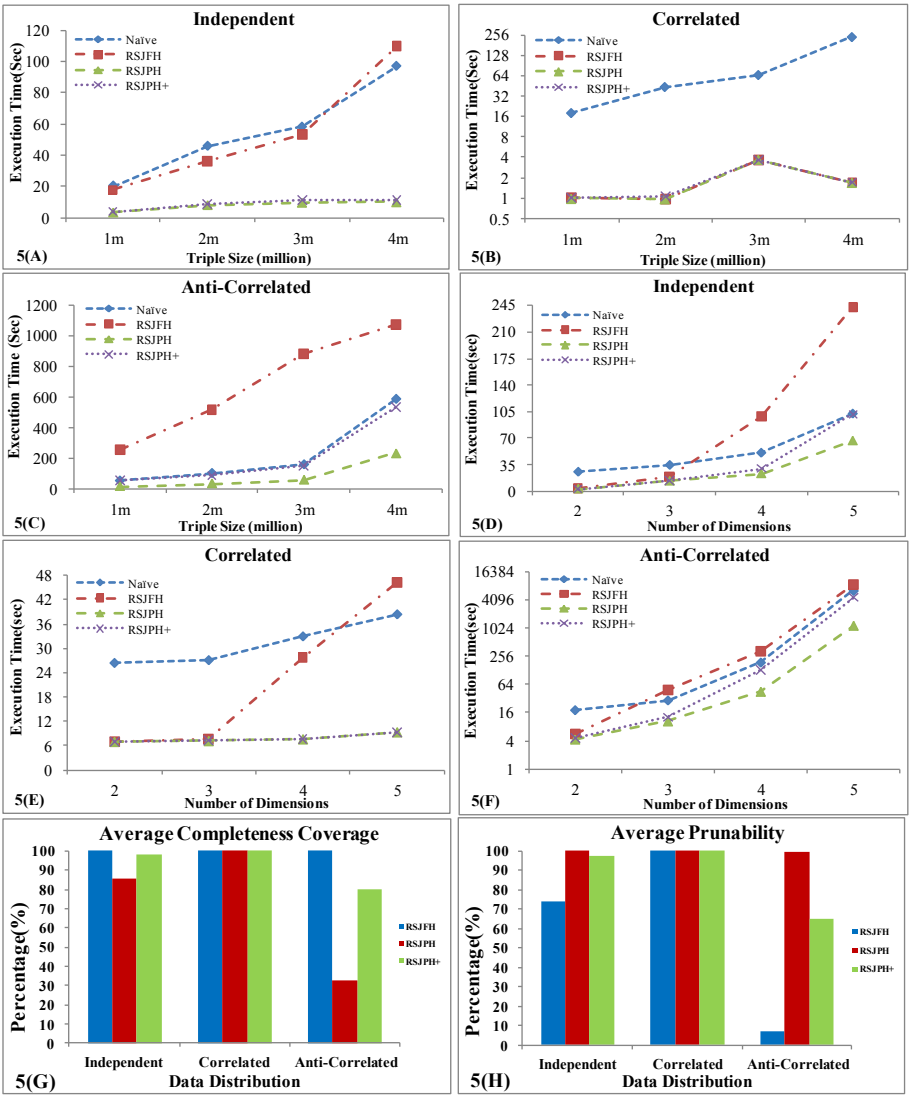


Fig. 5. Experimental Evaluation

Dimensionality. Figure 5 (D), (E) and (F) show the effect of increasing the dimensionality (2-5) on the performance of the four algorithms for different data distributions. As in previous experiments, *RSJPH* and *RSJPH+* consistently outperform *RSJFH* and *Naïve*. The execution time of *RSJFH* starts to increase with number of dimensions greater than 3. The reason is that the conservative way of updating header point makes the header points greatly reduce the pruning power in high dimensional data and the extra comparisons almost double the total execution time. For *RSJPH+*, with the increase in number of dimensions, the size of the saved tuples by the crosscheck condition increases, therefore, the size of candidate skyline set increases and the execution time increases as well.

Average Completeness Coverage and Average Prunability. Figure 5 (G) and (H) show the average completeness coverage (*ACC*) and average prunability (*AP*) of *RSJFH*, *RSJPH* and *RSJPH+*. *ACC* and *AP* are the averages for completeness coverage and the number of pruned triples across all the experiments shown in Figure 5 (A) to (F) respectively. The pruned triples include the ones pruned by header points as well as the ones pruned by early termination. Figure 5 (E) shows that *RSJFH* has 100% of *ACC* in all data distributions. For the correlated datasets, the data distribution is advantageous in forming a strong header point without harming the completeness of skyline results when the data is sorted from "best" to "worst". Thus, *RSJFH*, *RSJPH* and *RSJPH+* have 100% of *ACC* and 99% of *AP* in correlated datasets. For independent datasets, *RSJPH* aggressively updates the header points to increase *AP* with the cost of decreasing *ACC*. *RSJPH+* improves the *ACC* by using crosscheck while only sacrificing 2.7% of the *AP* compared with *RSJPH*. For anti-correlated datasets, the data distribution makes all the algorithms perform poorly. Although *RSJFH* achieves 100% of *ACC*, the *AP* decreases to 7%. *RSJPH* still maintains 99% for *AP* but its *ACC* is only 32%.

RSJPH+ achieves a good tradeoff between completeness coverage and prunability. *RSJPH+* computes about 80% of skyline results when it scans about the first 35% of sorted datasets.

6 Related Work

In recent years, much effort has been spent on evaluating skyline over single relation. [1] first introduced the skyline operator and proposed BNL and D&C and an algorithm using B-trees that adopted the first step of Fagin's A_0. [2][3][9] proposed algorithms that could terminate earlier based on sorting functions. [4][5][6] proposed index algorithms that could progressively report results. Since these approaches focus on single relation, they consider skyline computation independent from join phase, which renders the query execution to be blocking.

Some techniques have been proposed for skyline-join over multiple relations. [11] proposed a partitioning method that classified tuples into three types: general, local and non-local skyline tuples. The first two types are joined to generate a subset of the final results. However, this approach isn't suitable for single dimension tables like VPT [7] in RDF databases because each VPT can only be divided into general and local skyline tuples, neither of which can be pruned, requiring a complete join of all relevant tables. [16] proposed a framework SkyDB to partition the skyline-join

process into macro and micro level. Macro level generates abstractions while micro level populates regions that are not pruned by macro level. Since RDF databases only involve single dimension tables, SkyDB is not suitable for RDF databases.

In addition, there are some techniques proposed for skyline computation for Semantic Web data and services. [8] focused on extending SPARQL with support of expression of preference queries but it does not address the optimization of query processing. [13] formulated the problem of semantic web services selection using the notion of skyline query and proposed a solution for efficiently identifying the best match between requesters and providers. [14] computed the skyline QoS-based web service and [15] proposed several algorithms to retrieve the top-k dominating advertised web services. [17] presented methods for automatically relaxing over-constrained queries based on domain knowledge and user preferences.

7 Conclusion and Future Work

In this paper, we have addressed the problem of skyline queries over RDF databases. We presented the concept of *Header Point* and *Early Termination* to prune non-skyline tuples. We have proposed a complete algorithm *RSJFH* that utilized the prunability of *Header Point* and *Early Termination.* Then, we proposed two near-complete algorithms, *RSJPH* and *RSJPH+*, for achieving the tradeoffs between quick response time and completeness of skyline queries over RDF databases. In future, we will integrate cost-based techniques for further optimization. We will also address the issue of incomplete skyline computation over RDF databases.

Acknowledgement. The work presented in this paper is partially funded by NSF grant IIS- 0915865. Thanks to Mridu B. Narang for the draft comments and to Jigisha Dhawan, Vikas V. Deshpande, Amrita Paul and Gurleen Kaur for the discussions.

References

[1] Borzsonyi, S., Kossmann, D., Stocker, K., Passau, U.: The Skyline Operator. In: ICDE (2001)
[2] Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with Presorting. In: ICDE (2003)
[3] Bartolini, I., Ciaccia, P., Patella, M.: SaLSa: computing the skyline without scanning the whole sky. In: CIKM, Arlington, Virginia, USA, pp. 405–414 (2006)
[4] Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient Progressive Skyline Computation. In: VLDB, San Francisco, CA, USA, pp. 301–310 (2001)
[5] Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: an online algorithm for skyline queries. In: VLDB, HK, China, pp. 275–286 (2002)
[6] Papadias, D., Fu, G., Morgan Chase, J.P., Seeger, B.: Progressive Skyline Computation in Database Systems. ACM Trans. Database Syst (2005)
[7] Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB, Vienna, Austria, pp. 411–422 (2007)
[8] Siberski, W., Pan, J.Z., Thaden, U.: Querying the semantic web with preferences. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 612–624. Springer, Heidelberg (2006)

[9] Godfrey, P., Shipley, R., Gryz, J.: Maximal Vector Computation in Large Data Sets. In: VLDB, Norway (2005)
[10] Raghavan, V., Rundensteiner, E.A.: Progressive Result Generation for Multi-Criteria Decision Support Queries. In: ICDE (2010)
[11] Jin, W., Ester, M., Hu, Z., Han, J.: The Multi-Relational Skyline Operator. In: ICDE (2007)
[12] Sun, D., Wu, S., Li,J.,Tung, A.K.H.: Skyline-join in Distributed Databases. In: ICDE Workshops, pp. 176–181 (2008)
[13] Skoutas, D., Sacharidis, D., Simitsis, A., Sellis, T.: Serving the Sky: Discovering and Selecting Semantic Web Services through Dynamic Skyline Queries. In: ICSC, USA (2008)
[14] Alrifai, M., Skoutas, D., Risse, T.: Selecting Skyline Services for QoS-based Web Service Composition. In: WWW, Raleigh, NC, USA (2010)
[15] Skoutas, D., Sacharidis, D., Simitsis, A., Kantere, V., Sellis, T.: Top-k Dominant Web Services Under Multi-Criteria Matching. In: EDBT, Russia, pp. 898–909 (2009)
[16] Raghavan, V., Rundensteiner, E.: SkyDB: Skyline Aware Query Evaluation Framework. In: IDAR (2009)
[17] Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing RDF Queries based on User and Domain Preferences. JIIS 33(3) (2009)