Enabling a Package Query Paradigm on the Semantic Web: Model and Algorithms

Matthew Sessoms(✉) and Kemafor Anyanwu

Semantic Computing Research Lab, Department of Computer Science,
North Carolina State University, Raleigh, NC, USA
{mwsessom,kogan}@ncsu.edu
http://www.ncsu.edu

Abstract. The traditional search model of finding links on the Web is unsatisfactory for the increasingly complex tasks that seek to leverage the diverse, increasingly structured and semantically annotated data on the Web. A good example is when users seek to find collections or packages of resources that meet some constraints e.g., a collection of learning resources that cover some topics and have a good average rating or a collection of tourist attractions in a city such that total cost and total travel time for visiting all attractions meet the given constraints. For such queries, the goal is the return a set of constraint-qualifying collections or *packages*. However, using the traditional "set of links" query paradigm, such queries can only be satisfied by issuing multiple queries, reviewing answer lists and manually assembling packages to suit a user's desired constraints.

In this article, we introduce the concept of a *Package Query* for querying for resource combinations on the Semantic Web. In particular, we consider a frequent subclass of such queries *Skyline Package Queries*, in which multiple competing criteria are specified in the query so that the pareto-optimal set or skyline of packages are returned. In contrast to a few recent efforts on package queries on single relational models, fine-grained data models such as RDF include the challenge of computing the package skyline over multiple joins of ternary relations. We present four evaluation strategies involving different combinations of relational query operators and a new operator for Skyline Package Queries and different storage models for RDF data. A comparative evaluation of the algorithms over real world and synthetic-benchmark RDF datasets is presented.

Keywords: RDF · Package Skyline queries · Performance

1 Introduction

The Web has become a dominant knowledge source that informs a wide variety of technical and non-technical decisions. An increasing number of decision tasks being supported by Web data require more complex search paradigms than the

A. Hameurlain et al. (Eds.): TLDKS XIII, LNCS 8420, pp. 1–32, 2014.
DOI: 10.1007/978-3-642-54426-2_1,

mainstream "list of url matches" paradigm on the Web. Although there may be limitations to the degree of search complexity that is possible over unstructured content, the growing success of the Semantic Web offers the potential of leveraging its large amount of structured knowledge content for answering complex questions. An interesting querying use case that has broad applications is one where satisfying a user's query can be achieved only with combinations or "packages" of resources and not individual resources. Therefore, a query result is a list of resource combinations and not simply a list of resources. As a more concrete example, consider the following scenario.

Motivating Example (*e-learning*). A student would like to find e-learning resources from a collection of semantically annotated e-learning resources, e.g., [37], covering a set of topics on an upcoming test: relational model, algebra and calculus. Since creators of resources have the flexibility to modularize their content as they see fit, these topics may be covered by a single resource by one author, or by multiple resources by a different author, e.g., splitting content into two units: "data modeling" which covers the relational model, and then "querying" which covers relational algebra and calculus. Yet another content creator has a seperate content resource for each of the three topics. The implication is that in order to satisfy the user's query, some results, which match only portions of the answer, will need to be "packaged" into combinations that completely satisfy the query. Further, assuming that individual resources have ratings from reviews and maybe subscription charges, the user may have package preferences e.g., average rating resources of resource combination is maximized while total cost is minimized. It is not difficult to imagine many other applications of such queries. As another example consider a tourist interested in finding a number of tourist attractions at a destination where total distance traveling between them and total prices are minimized.

Since traditional querying models focus on finding list of items *rather than a list of item combinations*, the current strategy for arriving at such results is for users to review answer lists of items for possibly multiple queries and assemble and compare packages manually. In this article, we consider the idea of *skyline package queries over RDF data* which compute a pareto-optimal set of packages over RDF data, given multiple global or package preference criteria. There are three key fundamental challenges to be addressed: first is finding "relevant elementary or partial matches" i.e., items that meet *some* part of the description e.g., resources on just relational algebra, which can be combined with other partial matches to form complete results; second, is the number of joins required to stitch together the fine-grained (binary relations) representation of data in an RDF model to find matches; third is the computation of the skyline of combinations from elementary matches given package preferences. Depending on whether these tasks are done independently or holistically, query evaluation could be very expensive. The combination of these three challenges distinguish this from the few recent efforts on skyline [48] or top-k [23,44,45] package queries over single relations requiring no joins and in the latter case consider only a single preference criteria. While our previous work addresses the challenge of computing item

skylines over RDF data models, the introduction of the packaging requirement in package queries limits the possibility of adapting the previously proposed approach in an efficient manner. Other work on skylining over joins in relational models [32,43,47] consider very restricted join patterns (single join [43], star-like schemas [47]) that are not applicable to RDF and also do not consider packages.

In this article, we present an efficient algebraic interpretation of package skyline queries over RDF in terms of a special operator and storage model. We also consider alternative interpretations that rely mostly on existing query operators and the mainstream vertical partitioned storage model for RDF. Specifically, we contribute the following: (1) a formalization of the logical query operator, *SkyPackage*, for package skyline queries over an RDF data model, (2) two families of query processing algorithms for physical operator implementation based on the traditional vertical partitioned storage model and a novel storage model here called *target descriptive, target qualifying* (TDTQ), and (3) an evaluation of the four algorithms proposed over synthetic and real world data. Specifically, we extend a previous paper [36] by proposing a more efficient algorithm, *SkyPackage*, for solving the skyline package problem and provide a more rigorous evaluation on the four algorithms. The rest of this article is organized as follows. The background and formalization of our problem is given in Sect. 2. Section 3 introduces the two algorithms based on the vertical partitioned storage model, and Sect. 4 presents an algorithm based on our TDTQ storage model. An empirical evaluation study is reported in Sect. 5, and related work is described in Sect. 6. The article is concluded in Sect. 7.

2 Background and Problem Definition

Consider a scenario where stores publish Semantic Web-enabled catalogs over which users can search based on products and some preferences. For example, a customer wishes to purchase milk, eggs, and bread from any combination of stores as long as the combination (*package*) offers an overall minimized total cost and overall maximized store ratings. Figure 1 shows an example ontology and data about stores and their ratings, items sold by each store and their prices. Here, one possible package is *aee* (i.e., buying milk from store *a*, eggs from store *e* and bread also from store *e*). Another possibility is *bee* (i.e., milk from store *b*, eggs and bread from store *e* as in the previous case). The bottom right of Fig. 1 shows the total price and average rating for packages *aee*, *bee* and *bge*. We see that *aee* is a better package than *bee* because it has a smaller total price and the same average rating. On the other hand, *bge* and *aee* are incomparable because although *bge*'s total price is worse than *aee*'s, its average rating is better.

Problem Definition. To contextualize the interpretation of such queries on RDF data, we build its algebraic interpretation on top of existing operators for interpretating SPARQL graph pattern queries. Observe that the example description contains some key components: a base graph pattern structure (the SPARQL queries in the figure). Graph pattern queries have return clauses (the

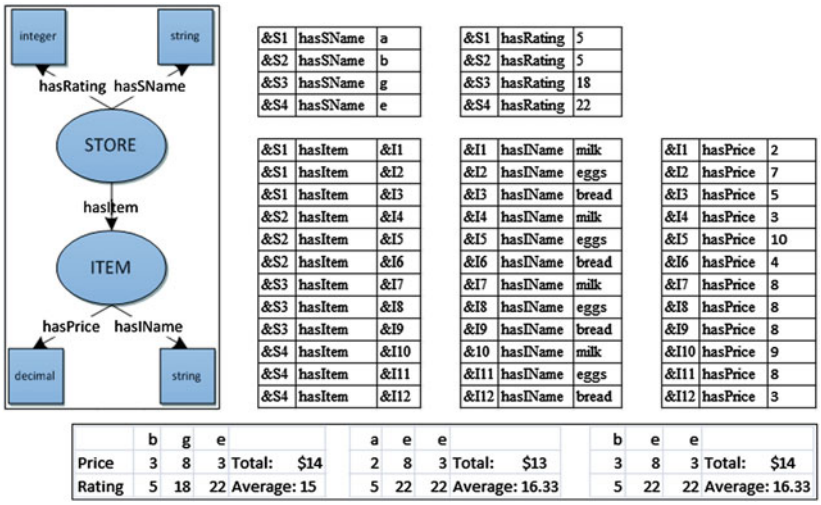


Fig. 1. Data For E-commerce example

SELECT clause) which denote the variables (*return variables*) whose bindings we are interested in including in the result. One of those variables, the (*target variable - ?store*) captures the *targets* of the query (*stores*) while the rest are part of the subquery structure that qualifies valid targets e.g., stores *should sell milk* and are called *target qualifing constraints - (?item, hasName, "milk").* There are analogous graph patterns for qualifiers "selling *bread* and*eggs*". The second component of the query specification defines combinations or *packages* which can interpreted as a Fig. 2 of the results of above queries. The third component is pruning packages based on *preferences*,e.g., *minimizing the total price of package.* It is important to note that the preferences here are specified over aggregates of datatype properties (i.e., attributes) that are part of the description for targets (e.g., maximizing the average over store ratings) or target qualifiers, e.g., minimizing total price. The base graph patterns and crossproduct operations can be expressed using SPARQL's algebra or high-level language. The expression of preferences and item skyline queries are not supported by SPARQL although there have been some proposals for algebraic extensions to the SPARQL algebra. Our goal here, is to extend the SPARQL algebra to include operators that allow the algebraic expression of package skyline queries. We do not address SPARQL language grammar extensions here. The new operators to be introduced can be seen as a generalization the item skyline operators which when combined with operators for expressing graph pattern queries can express package skyline queries. In other words, in our framework, traditional item skyline queries will be viewed as package skyline queries where the package size is 1.

More formally, let D be a dataset with property relations $P_1, P_2, \ldots, P_m$ and GP be a graph pattern with triple patterns $TP_i, TP_j, \ldots, TP_k$ (TP_x means triple

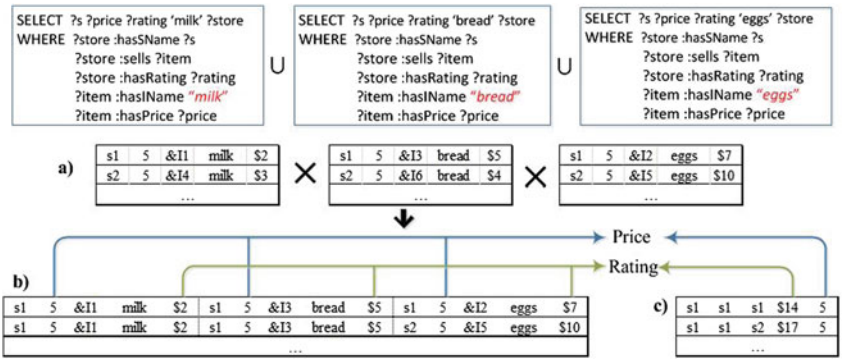


Fig. 2. Dataflow for the SkyPackage problem in terms of traditional query operators

pattern with property P_x). $[[\text{GP}]]_D$ denotes the answer relation for GP over D, i.e., $[[\text{GP}]]_D = P_i \bowtie P_j \bowtie \ldots \bowtie P_k$. Let $var(TP_x)$ and $var(GP)$ denote the variables in the triple and graph pattern respectively and $r \in var(GP)$ denote the target variable. Note that the rest of the graph pattern is used to define the target i.e target description and relationship between qualifiers and targets.

For preference specification, we begin by reviewing the formalization for preferences given in [32]. Let $Dom(a_1), \ldots, Dom(a_d)$ be the domains for the columns $a_1, \ldots, a_d$ in a d-dimensional tuple $t \in [[\text{GP}]]_D$. Given a set of attributes $B \subseteq A'$, a preference PF over the set of tuples $[[\text{GP}]]_D$ is then defined as $PF := (B; \prec_{PF})$, where $\prec_{PF}$ is a strict partial order on the domain of B. Given a set of preferences $PF_1, \ldots, PF_m$, their combined Pareto preference PF is defined as a set of equally important preferences.

For a set of d-dimensional tuples R and preference $P = (B; \prec_P)$ over R, a tuple $r_i \in R$ dominates tuple $r_j \in R$ based on the preference P (denoted as $r_i \prec_P r_j$), iff $(\forall(a_k \in B)(r_i[a_k] \preceq r_j[a_k]) \wedge \exists(a_l \in B)(r_i[a_l] \prec r_j[a_l]))$.

Definition 1 (Skyline Query). *To adapt preferences to graph patterns, we associate a preference with the property (assumed to be a datatype property) on whose object the preference is defined or the* preference property *e.g., the property* rating. *Let PF_i denote a preference with preference property as P_i. Then, for a graph pattern $GP = TP_1, \ldots, TP_m$ and a set of preferences $PF = PF_i, PF_j, \ldots, PF_k$, a* skyline query *$SKYLINE[[[GP]]_D, PF]$ returns the set of tuples from the answer of GP such that no other tuples dominate them with respect to PF and they do not dominate each other. The extension of the skyline operator to packages is based on two functions* Map *and* Generalized Projection.

Definition 2. *Let $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ be a set of k mapping functions such that each function $f_j(B)$ takes a subset of attributes $B \subseteq A$ of a tuple t, and returns a value x.*

Map $\hat{\mu}_{[\mathcal{F},\mathcal{X}]}$ (adapted from [32]) *applies a set of k mapping functions $\mathcal{F}$ and transforms each d-dimensional tuple t into a k-dimensional output tuple t'*

defined by the set of attributes $\mathcal{X} = \{x_1, x_2, \ldots, x_k\}$ *with* x_i *generated by the function* f_i *in* $\mathcal{F}$.

Generalized Projection $\prod_{colr_x, colr_y, colr_z, \hat{\mu}_{[\mathcal{F},\mathcal{X}]}}$ (R) *returns the relation* $R'(colr_x, colr_y, colr_z, \ldots, x_1, x_2, \ldots, x_k)$. *In other words, the generalized projection outputs a relation that appends the columns produced by the map function to the projection of* R *on the attributes listed, i.e.,* $colr_x, colr_y, colr_z$.

Definition 3 (SkyPackage Graph Pattern). *A* SkyPackage *graph pattern query is graph pattern* $GP_{[r,\{c_1,c_2,\ldots,c_N\},\mathcal{F}=\{f_1,f_2,\ldots,f_k\},\{PF_i,PF_j,\ldots,PF_k\}]}$ *such that:*

1. c_i *is a triple pattern specifying a target qualifying constraint.*
2. $GP_{\{c_1,c_2,\ldots,c_N\}}$ *is the set of graph patterns* $GP_{c_1}, GP_{c_2}, \ldots, GP_{c_N}$ *associated with target qualifying constraints* $c_1, \ldots, c_N$ *e.g., each of the three graph patterns in Fig. 1 represent a* GP_{c_i}.
3. r *is a set of return variables such that* $r_i \in r$ *implies* $r_i \in \text{var}(GP_i)$ *is called the target of the query, e.g., stores.*
4. PF_i *is the preference specified on the property* P_i *whose mapping function is* f_i *or more specifically,* $PF_i = (f_i(P_i); \prec_{PF_i})$.

The answer *to a* SkyPackage *graph pattern query* R_{SKY} *can be described algebraically as the result of a sequence of operators in the following manner:*

1. $R_{product} = [[GP_{c_1}]] \times [[GP_{c_2}]] \times \cdots \times [[GP_{c_N}]]$ *such that* $[[GP_{c_x}]]$ *is the result of evaluating the branch of the union query with constraint* c_x. *Figure 2b shows the partial result of the crossproduct of the three subqueries in* (a) *based on the 3 constraints on milk, bread and eggs.*
2. $R_{project} = \prod_{r_1, r_2, \ldots r_N, \hat{\mu}_{[\mathcal{F}=\{f_1,f_2,\ldots f_k\},\mathcal{X}=\{x_1,x_2,\ldots x_k\}]}}$ $(R_{product})$ *where*

   r_i *is the column for the target variable in subquery* i*'s result,* $f_1 : (domc_1(o_1) \times domc_2(o_1) \times \cdots \times domc_N(o_1)) \rightarrow \mathbb{R}$ *where* $domc_1(o_1)$ *is the domain of values for the column representing the object of* P_1*, e.g., column for object of hasPrice, in* $[[GP_{c_1}]]$. *The functions in our example would be* $total_{hasPrice}$, $average_{hasRating}$. *The output of this step is shown in Fig. 2c.*
3. $SKYLINE[R_{project}, \{PF_{P'_i}, PF_{P'_j}, \ldots, PF_{P'_k}\}]$ *such that* $PF_{P'_i}$ *is the preference defined on the aggregated columns produced by the map function (denoted by* P'_i*), e.g., minimizing total price.*

Our example in this model is $GP_{[r,X,\mathcal{F},PF]}$, where

- $r = ?store$
- $X = \{(?item, hasName, \text{“}milk\text{”}), (?item, hasName, \text{“}bread\text{”}), (?item, hasName, \text{“}eggs\text{”})\ \}$
- $\mathcal{F} = \{SP = sum()_{price}, SR = average()_{rating}\}$
- $PF = \{(SP; \prec_{min}), (SR; \prec_{max})\}$

2.1 Related Work

Although much research has been done in the area of traditional skyline queries, package skyline queries have not received a generous amount of attention. Our contribution is unique from previous work in that we provide algorithms whose (package) skyline result contains elements of cardinality greater than one. In this section, we present previous work related to our study of skyline packages. We begin with a historical perspective of how the skyline query came about and an overview of some of the earlier solutions proposed outside of a database context. Then we provide an overview of solutions within a database context and their correspondence to single-relations, multi-relations, and composite top-k queries.

The skyline query problem originally arose in the theory field in the 1960s, and the skyline set was coined as the *Pareto set*. This problem became known as the *maximal vector problem* [28,34], whose solution (e.g., skyline) is called *maximal vectors* [6] or *admissible points* [2], and is similar to the *contour problem* [31] and *convex hull problem*. Solutions to the maximal vector problem were proposed in [5,6,28]; however, these solutions cannot scale to large databases because they require all data to be in memory.

Single-relation Skyline Algorithms. The first proposed method of applying the maximal vector problem to databases was [10] and the term *skyline queries* was coined. Since then, the skyline query problem has often been referred to as a secondary/external storage version of the maximal vector problem [24]. [10] originally introduced and provided a block nested loops, divide-and-conquer, and B-tree-based algorithms. Later, [18] introduced a sort-filter-skyline algorithm that is based on the same intuition as BNL, but uses a monotone sorting function to allow for early termination. Unlike [10,18,41], which has to read the whole database at least once, index-based algorithms [26,33] allow one to access only a part.

Multi-relation Skyline Algorithms. All of the previous algorithms are designed to work on a single relation. As the Semantic Web matures and RDF data is populated, there has been an increase in research involving multi-relational skyline queries. When queries involve aggregations, multiple relations must be joined before any of the above techniques can be used. Implicitly, the first work that deals with the problem of skyline over multiple relations via joins is [27]. Given a query that joins two relations and filters the result using a WHERE clause, the authors propose a method to overcome empty results known as *query relaxation*, which relaxes the join selection thus making the query more flexible. Unlike our work, they do not focus on preference queries.

Vlachou et al. [43] proposed a novel algorithm called SFSJ (sort-first skyline-join) that computes the complete skyline. Given two relations, access to the two relations are alternated using a pulling strategy, known as adaptive pulling, that prioritizes each relation based on the number of mismatched join values. SFSJ takes advantage of its early termination condition, which gives rise to its performance, when the two regions from each relation meet a certain condition,

Although the algorithm has no limitations on the number of skyline attributes, it is limited by two relations.

Recently, [16] introduced three skyline algorithms that are based on the concept of a *header point*, which allows some nonskyline tuples to be discarded before proceeding to the skyline processing phase. Raghavan and Rundensteiner [32] introduced a sky-join operator that gives the join phase a small knowledge of a skyline. An index-based, non-join skyline algorithm was proposed in [48]. Khabbaz and Lakshmanan [23] proposes a framework for collaborative filtering using a variation of top-k. However, their set of results do not contain packages but single items.

Composite Top-k Algorithms. Up until now, very little research has been conducted in the area of package, or composite, queries. Such previous work mostly aims at providing composite results in a recommendation system [44,45]. Xie [45] uses top-k techniques to provide a composite recommendation for travel planning. Since finding packages is complex and time consuming, most have oriented their work towards *approximating* the desired packages [44]. Unlike our goal, which is to provide the user with *all* correct results, this approach limits the user from seeing the complete results. Top-k is useful when ranking objects is desired. However, top-k is prone to discard tuples that have a 'bad' value in one of the dimensions, whereas a skyline algorithm will include this object if it is not dominated.

3 Algorithms for Package Skyline Queries over Vertical Partitioned Tables

We present in this section two approaches: *Join, Cartesian Product, Skyline* (*JCPS*) and *RDFSkyJoinWithFullHeader-Cartesian Product, Skyline* (*RSJFH-CPS*), for solving the package skyline problem. These approaches assume data is stored in vertically partitioned tables (VPTs) [1].

3.1 *JCPS* Algorithm

The formulation of the package skyline problem suggests a relatively straightforward algorithm involving multiple *joins*, followed by a *Cartesian product* to produce all combinations, followed by a single-table *skyline* algorithm (e.g., block-nested loop), called $JCPS$.

Consider the VPTs $hasIName$, $hasSName$, $hasItem$, $hasPrice$, and $hasRating$ obtained from Fig. 1. Solving the skyline package problem using $JCPS$ involves the following steps. First, we join all tables and perform a Cartesian product twice on I to obtain all store packages of size 3, as shown in Fig. 3a. As the product is being computed, the *price* and *rating* attributes are aggregated, as shown in Fig. 3b. Afterwards, a single-table skyline algorithm is performed to discard all dominated packages with respect to total price and average rating.

Algorithm 1 contains the pseudocode for such an algorithm. Solving the skyline package problem using $JCPS$ requires all VPTs to be joined together

(line 2), denoted as I. To obtain all possible combinations (i.e., packages) of targets, multiple specialized Cartesian products are performed on I (lines 3–6). A modified Cartesian product, denoted as $\otimes$, is implemented as a subroutine to ensure no duplicate target constraints (e.g., milk) are inlcuded. Afterwards, *equivalent* skyline attributes are aggregated (lines 7–10). Equivalent skyline attributes, for example, of the e-commerce motivating example would be price and rating attributes. Aggregation for the price of milk, eggs, and bread would be performed to obtain a total price. Finally, line 11 applies a single-table skyline algorithm to remove all dominated packages.

Algorithm 1: JCPS

Input: $VPT_1, VPT_2, \ldots VPT_x$ containing skyline attributes $s_1, s_2, \ldots, s_y$, and an aggregation function $\mathcal{A}(T)$ on some table T
Output: Package Skyline $\mathcal{P}$

```
1: n ← package size
2: I ← VPT_1 ⋈ VPT_2 ⋈ ··· ⋈ VPT_x
3: I_2 ← I × I
4: for all i ∈ [1, n − 2] do
5:     I_2 ← I_2 ⊗_{i,col} I
6: end for
7: M_1 ← A(I_{2_{s_1}})
8: for all i ∈ [2, y] do
9:     M_i ← A(M_{(i−1)_{s_i}})
10: end for
11: P ← skyline(M_y)
12: return P
```

subroutine $\otimes(T_1, T_2, col, iteration)$

```
1: list ← ∅
2: for all t1 ∈ T_1 do
3:     for all t2 ∈ T_2 do
4:         list.add(T_2(col))
5:         for all i ∈ [1, iteration] do
6:             if ( !list.contains(T_1(iteration + col)) ) then
7:                 list.add(T_1(iteration + col))
8:             else
9:                 list ← ∅
10:                break
11:            end if
12:            T_3 ← T_1(col) ⋈ T_2(col)
13:        end for
14:        list ← ∅
15:    end for
16: end for
17: return T_3
```

The limitations of such an algorithm are fairly obvious. First, many unnecessary joins are performed. Furthermore, if the result of joins is large, the input to the Cartesian product operation will be very large even though it is likely

item	price	store	rating
milk	2	A	5
eggs	7	A	5
⋮			

(a) Cartesian Product of Join Result

item1	item2	item3	store1	store2	store3	total price	average rating
milk	eggs	bread	A	B	C	10	5
milk	eggs	bread	B	B	A	7	5
⋮							

(b) Cartesian Product Result

Fig. 3. Resulting tables of $JCPS$

that only a small fraction of the combinations produced will be relevant to the skyline. The exponential increase of tuples after the Cartesian product phase will result in a large number of tuple-pair comparisons while performing a skyline algorithm. In addition, duplicates will have to be eliminated. To gain better performance, it is crucial that some tuples be pruned before entering into the Cartesian product phase, which is discussed next.

3.2 *RSJFH-CPS* Algorithm

A pruning strategy that prunes the input size of the Cartesian product operation is crucial to achieving efficiency. One possibility is to exploit the following observation: *skyline packages can be made up of only target resources that are in the skyline result when only one constraint (e.g., milk) is considered* (note that a query with only one constraint is equivalent to an item skyline query).

Lemma 1. *Let $\rho = \{p_1 p_2 \dots p_n\}$ be a package of size n (i.e., containing n target resources), $\mathcal{P}$ be the set of all skyline packages, and $p'_1, p'_2, \dots, p'_n$ be other target resources with respect to a qualifying constraint $C_1, C_2, \dots C_n$. If $\rho \in \mathcal{P}$, then $p_m \preceq_{C_m} p'_m$ for all $1 \leq m \leq n$.*

Proof. Let $\rho' = \{p_1 p_2 \dots p'_n\}$, where $p_n \preceq_{C_n} p'_n$, and let $x_1, x_2, \dots, x_m$ be the preference attributes for p_n and p'_n. Since $p_n \preceq_{C_n} p'_n$, $p_n[x_j] \preceq p'_n[x_j]$ for some $1 \leq j \leq n$. Therefore, $\mathcal{A}_{1 \leq i \leq n}(p_i[x_j]) \preceq \mathcal{A}_{1 \leq i \leq n}(p'_i[x_j])$, where $\mathcal{A}$ is a monotonic aggregation function and $p'_i \in \rho'$. Since for any $1 \leq k \leq n$, where $k \neq j$, $\mathcal{A}_{1 \leq i \leq n}(p_i[x_k]) = \mathcal{A}_{1 \leq i \leq n}(p'_i[x_k])$. This implies that $\rho \preceq \rho'$. Thus, ρ' is not a skyline package. □

As an example, let $\rho = \{p_1 p_2\}$ and $\rho' = \{p_1 p'_2\}$ and x_1, x_2 be the preference attributes for p_1, p_2, p'_2. We define the attribute values as follows: $p_1 = (3, 4), p_2 = (3, 5)$, and $p'_2 = (4, 5)$. Assuming the lowest values are preferred, $p_2 \preceq p'_2$ and

$p_2[x_1] \preceq p'_2[x_1]$. Therefore, $\mathcal{A}_{1\leq i\leq 2}(p_i[x_1]) \preceq \mathcal{A}_{1\leq i\leq 2}(p'_i[x_1])$. In other words, $(p_1[x_1] + p_2[x_1]) \preceq (p_1[x_1] + p'_2[x_1])$, i.e., $(3 + 3 = 6 \preceq 7 = 3 + 4)$. Since all attribute values except $p'_2[x_1]$ remained unchanged, by definition of skyline we conclude $\rho \preceq \rho'$.

This lemma suggests that the skyline phase can be pushed ahead of the Cartesian product step as a way to prune the input of the *JCPS*. Even greater performance can be obtained by using a skyline-over-join algorithm, *RSJFH* [16], that combines the skyline and join phase together. *RSJFH* takes as input two VPTs sorted on the skyline attributes. We call this algorithm *RSJFH-CPS*. This lemma suggests that skylining can be done in a divide-and-conquer manner where a skyline phase is introduced for each constraint, e.g., milk, (requiring 3 phases for our example) to find all potential members of skyline packages which may then be fed to the Cartesian product operation.

Given the VPTs *hasIName, hasSName, hasItem, hasPrice*, and *hasRating* obtained from Fig. 1, solving the skyline package problem using *RSJFH-CPS* involves the following steps:

1. $I^2 \leftarrow hasSName \bowtie hasItem \bowtie hasRating$
2. For each target t (e.g., milk)
   (a) $I_t^1 \leftarrow \sigma_t(hasIName) \bowtie hasPrice$
   (b) $S_t \leftarrow RSJFH(I_t^1, I^2)$
3. Perform a Cartesian product on all tables resulting from step 2b
4. Aggregate the necessary attributes (e.g., price and rating)
5. Perform a single-table skyline algorithm

Figure 4a shows two tables, where the left one, for example, depicts step (a) for milk, and the right table represents I^2 from step 1. These two tables are sent as input to $RSJFH$, which outputs the table in Fig. 4b. These steps are done for each target, and so in our example, we have to repeat the steps for *eggs* and *bread*. After steps 1 and 2 are completed (yielding three tables, e.g., *milk*, *eggs*, and *bread*), a Cartesian product is performed on these tables, as shown in Fig. 4c, which produces a table similar to the one in Fig. 3b. Finally, a single-table skyline algorithm is performed to discard all dominated packages.

Algorithm 2 shows the pseudocode for *RSJFH-CPS* The main difference between *JCPS* and *RSJFH-CPS* appears in line 5–8. For each target, a *select* operation is done to obtain all like targets, which is then joined with another VPT containing a skyline attribute of the targets. This step produces a table for *each* target. After the remaining tables are joined, denoted as I^2 (line 4), each target table I_i^1 along with I^2 is sent as input to $RSJFH$ for a skyline-over-join operation. The resulting target tables undergo a Cartesian product phase (line 9) to produce all possible combinations, and then all equivalent attributes are aggregated (lines 10–12). Lastly, a single-table skyline algorithm is performed to discard non-skyline packages (line 13). Since a skyline phase is introduced early in the algorithm, the input size of the Cartesian product phase is decreased, which significantly improves execution time compared to *JCPS*.

hasIName ⋈		hasPrice	
&I1	milk	&I1	2
&I4	milk	&I4	3

⋈

hasSName	⋈ hasItem	⋈ hasRating			
&S1	A	&S1	&I1	&S1	5
&S1	A	&S1	&I2	&S1	5

(a) $RSJFH$ (skyline-over-join) for milk

item	price	store	rating
milk	2	A	5
milk	3	B	5
⋮			

(b) $RSJFH$'s result for milk

item	price	store	rating
milk	2	A	5
⋮			

⋈

item	price	store	rating
eggs	3	B	5
⋮			

⋈

item	price	store	rating
bread	5	A	4
⋮			

(c) Cartesian product on all targets (e.g., milk, eggs, and bread)

Fig. 4. Resulting tables of $RSJFH$

Algorithm 2: RSJFH-CPS

Input: $VPT_1, VPT_2, \ldots VPT_x$ containing skyline attributes $s_1, s_2, \ldots, s_y$, and corresponding aggregation functions $\mathcal{A}_{s_1}(T), \mathcal{A}_{s_2}(T), \ldots, \mathcal{A}_{s_y}(T)$ on table T
Output: Package Skyline $\mathcal{P}$

$n \leftarrow$ package size
$t_1, t_2, \ldots, t_n \leftarrow$ targets of the package
VPT_1 contains targets and VPT_2 contains a skyline attribute of the targets
$I^2 \leftarrow VPT_3 \bowtie \cdots \bowtie VPT_x$
for all $i \in [1, n]$ **do**
    $I_i^1 \leftarrow \sigma_{t_i}(VPT_1) \bowtie VPT_2$
    $S_i \leftarrow RSJFH(I_i^1, I^2)$
end for
$T \leftarrow S_1 \times S_2 \times \cdots \times S_n$
$M_1 \leftarrow \mathcal{A}(T_{s_1})$
for all $i \in [2, y]$ **do**
    $M_i \leftarrow \mathcal{A}(M_{(i-1)_{s_i}})$
end for
$\mathcal{P} \leftarrow$ skyline(M_y)
return $\mathcal{P}$

MILK	
store	price
a	2
b	3
f	3
e	4
g	6
e	9
h	9
d	10

EGGS	
store	price
g	5
a	7
c	8
e	8
h	9
b	10
d	10

BREAD	
store	price
e	3
b	4
a	5
i	8
g	5
f	6
h	6
d	10

RATING	
store	value
e	5
i	5
c	12
f	13
h	14
g	18
a	20
d	21
b	22

Fig. 5. Target qualifying (*milk, eggs, bread*) and target descriptive (*rating*) tables for E-commerce example

4 Algorithms for Package Skyline Queries over the TDTQ Storage Model

4.1 The TDTQ Storage Model

While the previous two approaches, *JCPS* and *RSJFH-CPS*, rely on VPTs, the next approach is a multistage approach in which the first phase is analogous to the build phase of a hash-join. In our approach, we construct two types of tables: *target qualifying* tables and *target descriptive* tables, called $TDTQ$. Target qualifying tables are constructed from the target qualifying triple patterns (?*item hasIName* "*milk*") and the triple patterns that associate them with the targets (?*store sells ?item*). In addition to these two types of triple patterns, a triple pattern that describes the target qualifier that is associated with a preference is also used to derive the target qualifying table. In summary, these three types of triple patterns are joined per given constraint and a table with the target and preference attribute columns are produced. The left three tables in Fig. 5 show the the target qualifying tables for our example (one for each constraint). The target descriptive tables are constructed per target attribute that is associated with a preference, in our example *rating* for stores. These tables are constructed by joining the triple patterns linking the required attributes and produce a combination of attributes and preference attributes (store name and store rating produced by joining hasRating and hasSName). The rightmost table in Fig. 5 shows the target descriptive table for our example.

We begin by giving some notations that will aid in understanding of the TDTQ storage model. In general, the build phase produces a set of partitioned tables $T_1, \ldots, T_n, T_{n+1}, \ldots, T_m$, where each table T_i consists of two attributes, denoted by T_i^1 and T_i^2. We omit the subscript if the context is understood or if the identification of the table is irrelevant. $T_1, \ldots, T_n$ are the target qualifying tables where n is the number of qualifying constraints. $T_{n+1}, \ldots, T_m$ are the target descriptive tables, where $m-(n+1)+1 = m-n$ is the number of target attributes involved in the preference conditions.

4.2 *CPJS* and *SkyJCPS* Algorithms

Given the TDTQ storage model presented previously, one option for computing the package skyline would be to perform a Cartesian product on the target qualifying tables, and then joining the result with the target descriptive tables. We call this approach $CPJS$ (*Cartesian product, Join, Skyline*), which results in exponential time and space complexity. Given n targets and m target qualifiers, n^m possible combinations exist as an intermediate result prior to performing a skyline algorithm. Each of these combinations is needed since we are looking for a set of packages rather than a set of points. Depending on the preferences given, additional computations such as aggregations are required to be computed at query time. Our objective is to find all package skylines *efficiently* by eliminating unwanted tuples before we perform a Cartesian product. Algorithm 3 shows the $CPJS$ algorithm for determining package skylines.

Algorithm 3: CPJS

Input: $T_1, T_2, \ldots T_n, T_{n+1}, \ldots, T_m$
Output: Package Skyline $\mathcal{P}$
1: $I \leftarrow T_1 \times T_2 \times \cdots \times T_n$
2: **for all** $i \in [n+1, m]$ **do**
3: $\quad I \leftarrow I \bowtie T_i$
4: **end for**
5: $\mathcal{P} \leftarrow$ skyline(I)
6: **return** $\mathcal{P}$

CPJS begins by finding all combinations of targets by performing a Cartesian product on the target qualifying tables (line 1). This resulting table is then joined with each target descriptive table, yielding a single table (line 3). Finally, a single-table skyline algorithm is performed to eliminate dominated packages (line 5).

Applying this approach to the data in Fig. 5, one would have to compute all 448 possible combinations before performing a skyline algorithm. The number of combinations produced from the Cartesian product phase can be reduced by initially introducing a skyline phase on each target, e.g., milk, as we did in for *RSJFH-CPS*. We call this algorithm $SkyJCPS$. Although similarities to *RSJFH-CPS* can be observed, $SkyJCPS$ yields better performance due to the reduced number of joins. Figure 4a clearly illustrates that *RSJFH-CPS* requires four joins before an initial skyline algorithm can be performed. All but one of these joins can be eliminated by using the TDTQ storage model. To illustrate $SkyJCPS$, given the TDTQ tables in Fig. 5, solving the skyline package problem involves the following steps:

1. For each target qualifying table TQ_i (e.g., milk)
   (a) $I_{TQ_i} \leftarrow (TQ_i) \bowtie rating$
   (b) $I'_{TQ_i} \leftarrow$ skyline(I_{TQ_i})
2. $CPJS(I'_{TQ_1}, \ldots, I'_{TQ_i}, rating)$

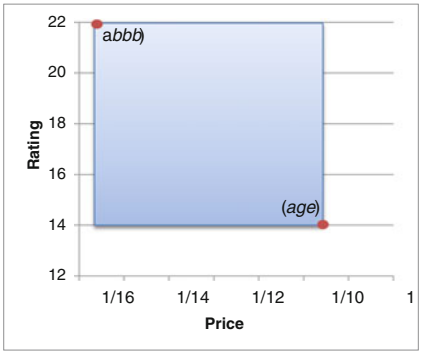


Fig. 6. Skyline region

Since the dominating cost of answering skyline package queries is the Cartesian product phase, the input size of the Cartesian product can be reduced by performing a single-table skyline algorithm over each target.

4.3 SkyPackage Algorithm

Our algorithm attempts to discern the skyline set by stepping through sorted sub-lists in a way that guarantees we move towards the skyline set. This strategy relies on being able to discover the best package in one dimension, which means we are guaranteed that no future packages will be dominated by this one. From Fig. 5, the package consisting of the first tuple from each of the target qualifying tables (*age*) constitutes a package skyline. Also, the package that has the best rating can be found by looking at the rating table for the highest rating, which is *b*. Thus, package (*bbb*) is a package skyline. To illustrate this, consider the two packages in Fig. 6, {(*age*),$10,14} and {(*bbb*), $17,22}. All future package skylines must fall between these two packages. We depict this in the shaded area. Any package that does not lie within this region can be immediately discarded. However, *candidate* packages may fall within this area and will need to be checked for membership in the skyline package.

Pruning and Early Termination. Since performing a Cartesian product is expensive and its output size to the number of package skylines ratio is very large, it is desirable to decrease this intermediate result. Therefore, we have to eliminate targets that cannot possibly be in the skyline set when packaged with any other available targets. While the naive approach would have to compose the packages and then perform a skyline to filter out unwanted tuples, our pruning technique offers *local prunability* that prunes tuples, i.e., targets, from individual target qualifiers before we form the packages.

To illustrate the concept of local pruning, consider the *milk* and *rating* tables in Fig. 5. We present in Fig. 7 four iterations where each iteration indicates a new tuple being examined. As we look at each store in the *milk* table, we probe

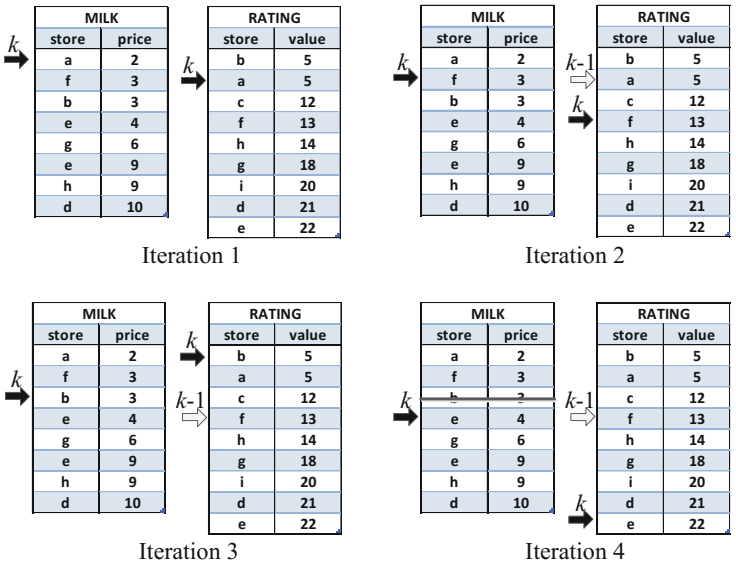


Fig. 7. Pruning example

the *rating* table and keep a pointer at its value. The first tuple examined, i.e., first iteration, in the *milk* table is $(a, 2)$. We probe the *rating* table to locate store a, and keep a pointer there for the next iteration(s). In the second iteration, we examine the next tuple $(f, 3)$ and probe the *rating* table again. We compare its value against the previous pointer, denoted as $k-1$. Since the current store is better than the previous store, we remove the pointer from store a and continue to the third iteration. As usual, the current tuple $(b, 3)$ is used to probe the *rating* table. In this case, the current pointer k $(b, 5)$ is worse than the previous pointer $k-1$ $(f, 13)$, and thus we prune the tuple $(b, 3)$. Because the current pointer that points to $(f, 13)$ is no better than the previous pointer, we save this pointer and examine the next tuple as shown in iteration 4.

The concept of local prunability is formalized in the following lemma.

Lemma 2 (Pruning). *Let $T_j[k]$ be the value of object k in table T_j, then $\forall k \in {T_i}^1, i \in [1, n]$, if $\exists j \in [n+1, m]$ such that $T_j[k-1] \prec T_j[k]$, and $T_i[k-1] \preceq T_i[k]$ then object k does not produce a package skyline and can be pruned from T_i.*

Proof. Since T_i is sorted in the preprocessing phase of our database, we know $T_i[k-1] \preceq T_i[k]$. Assume that k produces a package skyline (is part of the combination). Then, if $T_j[k] \preceq T_j[k-1]$, object $k-1$ dominates k, thus k cannot be part of the package skyline. □

Lemma 2 ensures that any combination where k appears is not a package skyline. If we denote the size of each table T as $|T|$, then for each tuple pruned in T_i, the size of the resulting Cartesian product is reduce to $|T_1| \times |T_2| \times \cdots \times$

MILK	
store	price
a	2
b	3

EGGS	
store	price
g	5
a	7
b	10

BREAD	
store	price
e	3
b	4

Fig. 8. Pruning and early termination result

package	price	rating
age	10	14.3
agb	11	20
bgb	12	20.7
bab	14	21.3
bbb	17	22

Fig. 9. Skyline package result

$(|T_i| - 1) \times \cdots \times |T_n|$. To illustrate this, after tuple $(b, 3)$ is pruned in Fig. 4, the Cartesian product size is reduced from 448 to 392 tuples.

We define the *previous object* to be a pointer to the best last seen object and the pointer is updated when a better object is examined. In Lemma 2, we denote the current object as k and the previous object as $k - 1$. If Lemma 2 is not satisfied, the pointer that once pointed to $k - 1$ is updated to point to k.

To increase performance of our algorithm, we utilize the following early termination strategy for each resource.

Lemma 3 (Early Termination). *If k is the current object in T_i and for all $j \in [n + 1, m], T_j[k]$ is the best in T_j, then stop examining T_i and continue to T_{i+1}, $i + 1 \leq n$.*

Proof. Assume there exists an object $k + 1$ in T_i that has not been examined. Then $T_i[k] \preceq T_i[k + 1]$, and $T_j[k] \preceq T_j[k + 1]$. If $T_j[k + 1] \neq T_j[k]$, then $T_j[k] \prec T_j[k+1]$. Then for any object after k, $T_i[k] \preceq T_i[k+1] \preceq T_i[k+2] \cdots$. Therefore, every object after k is dominated. □

Lemma 3 allows us to stop examining tuples in a given table when the best object is seen in the target descriptive tables , i.e., *rating* table. For example, in Fig. 7, since b has the highest rating, we stop scanning the *milk* table once b is examined and prune all tuples below it. If a target qualifying table does not contain the best object from the target descriptive table, we choose the next best object such that the target qualifying table contains this object.

After pruning, our next phase is performing a Cartesian product. As the product is produced, if there exist any tuples that do not satisfy (local) hard constraints, we discard these. Figure 8 shows the resulting tables after pruning.

After the pruning phase is complete, a Cartesian product is performed among the target qualifying tables and joined with the target descriptive table(s) for

aggregation. In Fig. 8, a Cartesian product involving the *milk*, *eggs*, and *bread* tables is performed to find (1) all packages and (2) total price. The intermediate result is then joined with the *rating* table to find the average rating. If there exists any tuples that do not satisfy (global) hard constraints, we discard these. A skyline algorithm is performed to remove any packages that are not skyline packages. The final skyline package set is shown in Fig. 9.

Discussion. Now that we have provided a concrete example of the SkyPackage algorithm, we will now explain the pseudocode in Algorithm 4. Lines 1–3 of the algorithm explain some notations that are used within the algorithm. Once the query is issued, we examine each of the n tables (line 4) one row at time (line 6), keeping a pointer p that points to the $n+1$ table that has the best value. With each iteration, we initialize *ptr* (line 5) to the first object in T_i mapped to T_{n+1}. Then we check whether Lemma 2 holds (line 7). Lines 8–14 handles the case when two consecutive objects have the same value in T_i. In this case, the tuple with the worse value in T_{n+1} is pruned. Lines 15–17 are similar to lines 8–14 except the equality checks are done on T_{n+1} rather than T_i. That is if two objects have the same value in T_{n+1}, we prune the tuple that has the smallest value in T_i. In line 18, we reach our early termination check, Lemma 3. We can safely stop examining the current table when we access an object that has the lowest value in t_{n+1}. It can easily be showed that any tuple after this one cannot be in the skyline set. At this point, *ptr* can no longer be updated since any subsequent tuple will have a higher value in t_{n+1}. If local constraints are given, we perform a check in line 19 to determine whether the current tuple satisfies the constraints. If the current tuple is not satisfied, all tuples below and including this one are pruned. We then join the tables, removing any tuples that do not satisfy any global constraints. Lastly, any known skyline algorithm is performed.

5 Sesame Integration Framework

5.1 Sesame

Sesame [14] is an open-source RDF database implemented in Java whose architecture allows for persistent storage of RDF data and querying of that data. We chose Sesame as our RDF engine for a number of reasons. First, since Sesame is a server-based application, it allowed us to store and query data on the Semantic Web remotely. Second, Sesame does not require a specific communication protocol or storage mechanism to be used.

5.2 Framework

The data that was used in the framework was the MovieLens[1] dataset, which was converted to RDF format using the Jena API [15]. Its ontology is depicted in Fig. 10. We used a server with Linux and an Apache Tomcat Web container.

[1] http://www.grouplens.org/node/73/

Algorithm 4: SkyPackage

Input: $T_1, T_2, \ldots T_n, T_{n+1}, \ldots, T_m$
Output: Package Skyline $\mathcal{P}$

```
1: v_k(i) ← the value of object k in table i
2: k ← the current object (i.e., row)
3: ptr ← the best object
4: for all i ∈ [1, n] do
5:    ptr ← v_x(n + 1), x ← first tuple in i
6:    for all k ∈ t_i do
7:       check whether Lemma 2 holds
8:       if v_k(i) = v_{k−1}(i) then
9:          if  v_k(n + 1) > v_{k−1}(n + 1)  then
10:            prune (k, v_k(i))
11:         else
12:            prune (k − 1, v_{k−1}(i))
13:         end if
14:      end if
15:      if  v_k(n + 1) = v_{k−1}(n + 1) then
16:         prune max {(k, v_k(i)), (k − 1, v_{k−1}(i))}
17:      end if
18:      check whether Lemma 3 holds
19:      check k against local constraints
20:   end for
21: end for
22: cross product, remove tuples not satisfying global constraints
23: skyline
```

A Web-based interface was designed to allow users to query a subset of the MovieLens dataset. Although any package-related query can be supported, for the purpose of this article, we chose to support a query of the following form.

Query 1. *Given n movie-raters, find packages of n movies such that the average rating of all the movies is high and each movie-rater has rated at least one of the movies.*

When the user provides preferences using the Web-based interface, the information is sent to the SPARQL adapter that dynamically creates a SPARQL query. After the SPARQL query is executed and results of this query is available, the *SkyPackage* algorithm finds and presents the skyline package(s) that meet the user's preferences.

Data Storage. One option of storing RDF triples is to store them in a text file. However, this is inefficient for large numbers of triples and a solution involving indexing (e.g., database management system) is more appropriate. Relational databases, such as MySQL and Oracle, can be used to store such data, but are usually not optimized for such. Databases that are optimized for storage of RDF triples are called *triplestores*.

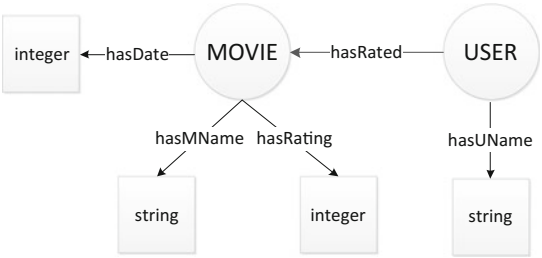


Fig. 10. MovieLens ontology

Sesame triplestore stores RDF triples in a *repository*. Sesame abstracts from any particular storage mechanism allowing a variety of repositories to be handled, including RDF triplestores and relational databases. Sesame offers several repositories in which to store data and all differs in where the data is stored. Two popular repositories are *memory* store and *native* store, corresponding to storage in-memory and on-disk, respectively. We used the native store configuration in our framework since it offers a better scalability solution for larger data sets as it is not limited to size of available memory. For native store, Sesame provides B-tree indexes on any combination of the subjects, properties, and objects. The index key(s) consist of subject (s), predicate (p), object (o), and context (c). The order in which these fields are listed determines the usability of the index. We chose to have as the index keys: *spoc* and *opsc*.

Data Retrieval. In order to retrieve data from Sesame's repository, we devised a skyline package operator, whose ultimate goal is to form queries that when executed will construct the TQ and TD tables. A description is given on how the queries for the TQ tables are constructed, followed by a similar description on how to construct the TD table.

We define *SP(*C, A, *PF)* as the skyline package operator that takes as arguments a list of constraints C, a list of properties A, and a list of preferences PF. In addition, we assume that the following VPTs exist: $vpt_1, vpt_2, \ldots, vpt_m$, $vpt_{m+1}, \ldots, vpt_n$. Moreover, for the purpose of illustration, we assume $vpt_1, \ldots$, vpt_m and $vpt_{m+1}, \ldots vpt_n$ are sufficient to construct the TQ and TD tables, respectively. The arguments for the SP operator are defined as follows:

- $C = (c_1, c_2, \ldots, c_m)$, where c_i are target qualifying constraints
- $A = (A_1 = (a_1, a_2), A_2 = (a'_1, a'_2))$, where a_i and a'_i are variables

A query is constructed for each target qualifying constraint (i.e., m queries) where the SELECT clause is formed by using variables in A_1 (e.g., SELECT $?a_1 ?a_2$). Within each query, the constraint $c_i \in C$ can be mapped to a FILTER clause or to a constraint in a WHERE clause of a SPARQL query. In order to map the constraints to a WHERE clause, a target qualifying triple pattern is constructed for each constraint. Assuming vpt_1 contains data to which a constraint can be applied, the target qualifying triple pattern is specified as ($?var$ $:vpt_1$ c_i),

```
SELECT ?movieName ?rating
WHERE { ?user hasName: "user8".
        ?user hasRated: ?movie.
        ?movie hasMName: ?movieName.
        ?movie hasRating: ?rating. }
```

```
SELECT ?movieName ?rating
WHERE { ?user hasName: "user34".
        ?user hasRated: ?movie.
        ?movie hasMName: ?movieName.
        ?movie hasRating: ?rating. }
```

(a) TQ Queries

```
SELECT ?movieName ?date
WHERE  {  ?movie hasName: ?movieName.
          ?movie hasDate: ?date. }
```

(b) TD Query

Fig. 11. Queries used to construct TD and TQ tables

where ?*var* is some variable. Moreover, the remaining tables $vpt_2 \dots vpt_m$ are joined together and with the intermediate result of the target qualifying triple pattern. A similar method can be applied if a FILTER clause is preferred. Instead of providing a constraint in the target qualifying triple pattern, a new variable is introduced, as in (?*var* :vpt_1 ?*constraint*). This constraint variable is then used in the FILTER clause along with the actual constraint to filter out unwanted results. An example FILTER clause is `FILTER regex(?constraint,` c_1).

To illustrate this process, consider Query 1 and the ontology depicted in Fig. 18. Suppose the person issuing the query is interested in the following movie-raters: *user*8 and *user*34. Since the query is requesting movies (i.e., the name of the movies) whose rating is maximized, we define $A = (?movieName, ?rating)$ because rating depends on the movie and the movie-rater. In addition, by examining Fig. 18, we have the following VPTs: *hasName, hasRated, hasMName, hasRating*. Since vpt_1 is $hasName$, we apply each constraint to this table by using a target qualifying triple pattern, such as (?*user* $hasName$ "user8"). Therefore, given C and A_1, the two queries in Fig. 12a are constructed, which produces the TQ tables.

A similiar approach is used to form the TD table. Since no target qualifying constraints are needed in this case, no FILTER condition is required and the only argument of interest is A_2, which contains variables that will be listed in the SELECT clause. The WHERE claused is formed by joining $vpt_{m+1}, \dots vpt_n$. Continuing from the previous example, we have the following VPTs: *hasName, hasDate*, and $A_2 = (?movieName, ?date)$. Therefore, the query in Fig. 12b is constructed.

In order to retrieve data from Sesame's repository, we implemented a server-side *SPARQL adapter* whose primary task is issuing SPARQL queries and serves as an intermediary between Sesame and *SkyPackage*. The adapter utilizes Sesame's *HTTPRepsoitory* component to execute the three queries and to retrieve the results. The results of each query are then stored in a data structure for processing and sent to *SkyPackage* along with the list of preferences PF. After *SkyPackage* is performed on the data returned from the adapter, the results are presented to the user. Figure 11 provides a high-level overview indicating the main components of our framework.

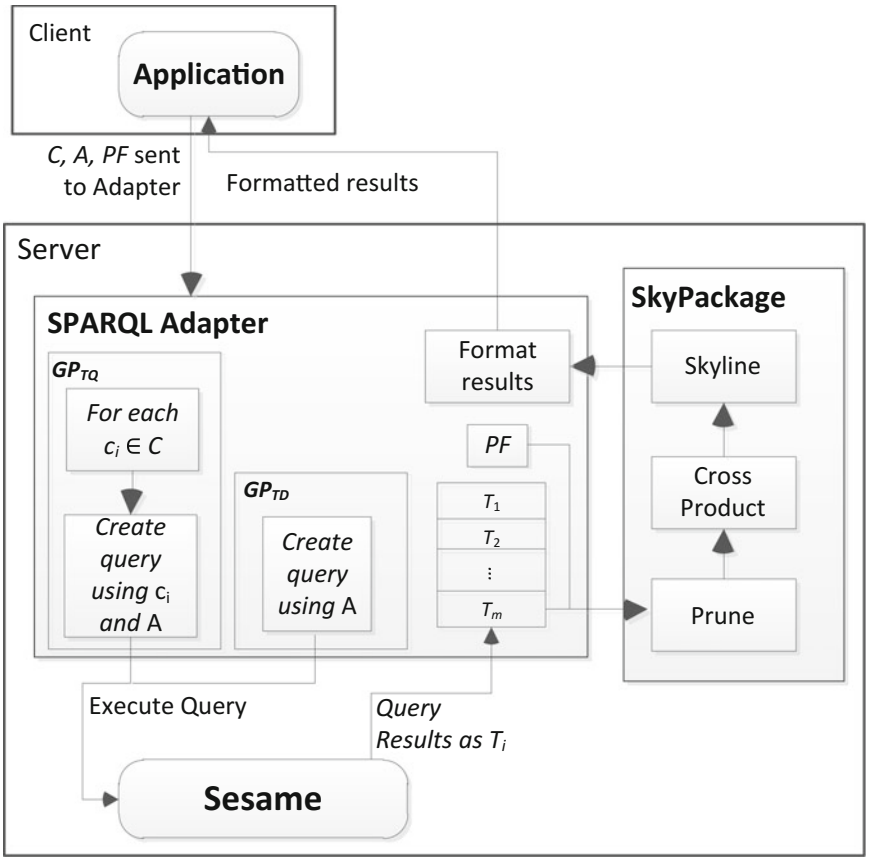


Fig. 12. Framework with SPARQL adapter

6 Evaluation

The main goal of this evaluation was to compare the performance of the proposed algorithms using both synthetic and real datasets with respect to package size scalability. In addition we compared the the feasibility of answering the skyline package problem using the VPT storage model and the TDTQ storage model.

6.1 Setup

All experiments were conducted on a Linux machine with a 2.33GHz Intel Xeon processor and 40GB memory, and all algorithms were implemented in Java SE 6. All data used was converted to RDF format using the Jena API and stored in Oracle Berkeley DB.

We compared four algorithms, *JCPS*, *SkyJCPS*, *RSJFH-CPS*, and *SkyPackage*. During the skyline phase of each of these algorithms, we used

the *block-nested-loops* (BNL) [10] algorithm. The package size metric was used for the scalability study of the algorithms. Since the Cartesian product phase is likely to be the dominant cost in skyline package queries, it is important to analyze how the algorithms perform when the package size grows, which increases the input size of the Cartesian product phase.

6.2 Synthetic Data

Dataset. Since we are unaware of any RDF data generators that allow generation of different data distributions, the data used in the evaluations were generated using a synthetic data generator [2]. The data generator produces relational data in different distributions, which was converted to RDF using the Jena API. We generated three types of data distributions: correlated, anti-correlated, and normal distributions. For each type of data distribution, we generated datasets of different sizes and dimensions.

Data Size Scalability. The first evaluation was performed to compare execution time among the three algorithms within the same package size using the three data distributions. The data consisted of triple sizes ranging from 450 to 635 and package sizes ranging from 2 to 5. While this may appear to be orders of magnitudes smaller than traditional evaluation corpora, it is important to note that the search space for package queries grows more aggressively than that of traditional pattern matching. We chose this triple size range to ensure that packages of different sizes can easily be compared and also to ensure that evaluation results would come in a reasonable time for larger package sizes. An increase in package size implies an increase in the number of tables, which also implies more Cartesian products. To ensure that the triple size remained approximately the same across different package sizes, we reduced the number of tuples in each table as the package size increased. Figure 13 shows the triple size and the number of tuples in each table for each package size as well as the approximate Cartesian product size.

While a triple size of 635 may seem small, Fig. 13 indicates that this triple size yields approximately 52.5 M tuples for a package size of 5 after a Cartesian product is performed. We were unable to obtain any results for triple size 635 using a package size of 5 for $JCPS$, as it ran for hours on this dataset. Figures 14 and 15 show the results and are plotted using a logarithmic scale. No anomalies were found within packages of the same size.

Package Size Scalability. In Fig. 16, we show how the algorithms perform across packages of size 2 to 5 for the a triple size of approximately 635 triples. Due to the exponential increase of the Cartesian product phase, this triple size is the largest possible in order to evaluate all three algorithms. For all package sizes, $SkyJCPS$ performs better than $JCPS$ because of the initial skyline algorithm performed to reduce the input size of the Cartesian product phase. $RSJFH$ outperformed $SkyJCPS$ for packages of size 5. We argue that $SkyJCPS$ may perform slightly slower than $RSJFH$ on small datasets distributed among many tables. In this scenario, $SkyJCPS$ has six tables to examine, while $RSJFH$ has

Package Size	Triple Size	Tuples in each table	Approx Cartesian Product Size
5	635	35	52.5M
	599	33	39M
	545	30	24M
	509	28	17M
	455	25	9.7M
4	634	42	3.1M
	604	40	2.5M
	544	36	1.7M
	499	33	1.2M
	454	30	810,000

Package Size	Triple Size	Tuples in each table	Approx Cartesian Product Size
3	639	53	149,000
	603	50	125,000
	543	45	91,000
	507	42	74,000
	447	37	50,000
2	632	70	4,900
	605	67	4,500
	542	60	3,600
	497	55	3,000
	452	50	2,500

Fig. 13. Synthetic data triple sizes

Fig. 14. Evaluation results for package sizes 2 and 3

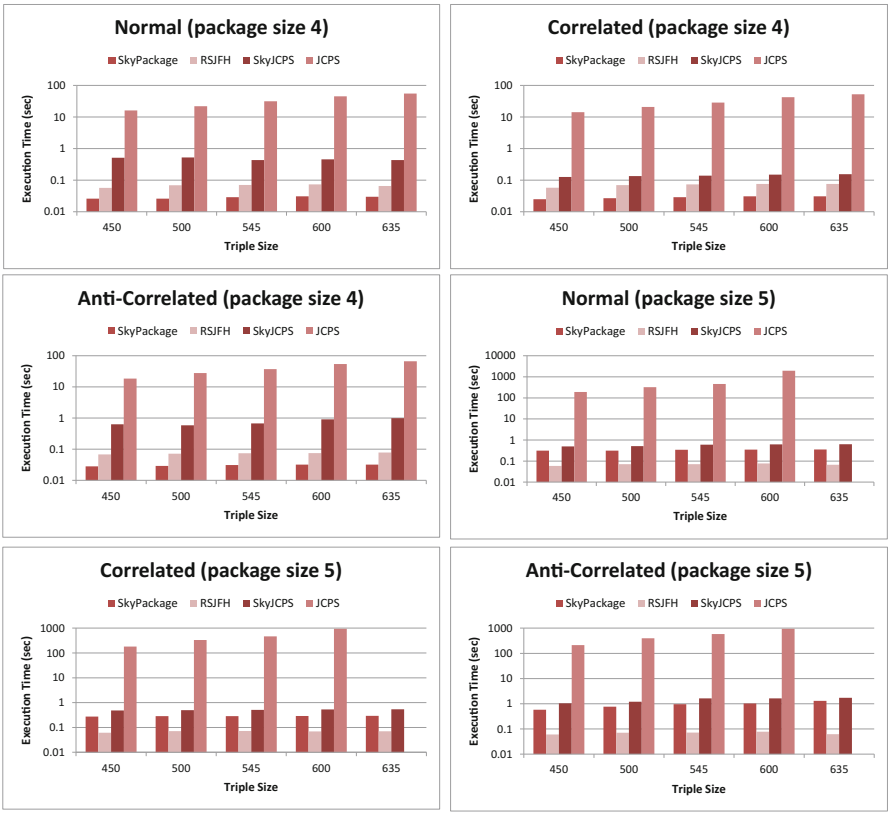


Fig. 15. Evaluation results for package sizes 4 and 5

only two tables. Evaluation results from the real datasets, which is discussed next, ensure us that *SkyJCPS* significantly outperforms *RSJFH* when the dataset is large and distributed among many tables. Due to the logarithmic scale used, it may seem that some of the algorithms have the same execution time for equal sized packages. This is not the case, and since *BNL* was the single-table skyline algorithm used, the algorithms performed best using correlated data and worst using anti-correlated data.

Average Prunability. To evaluate *SkyPackage*'s prunability, we collected the number of tuples that entered the Cartesian product phase and compared it to the total number of initial tuples for each data distribution and triple size. We then took the average over the three data distributions. The average prunability results can be seen in Fig. 17. As the package size increases, a larger percentage of tuples enters the Cartesian product phase. Even though the triple size remains approximately the same in all packages, the total number of tuples increases as the package size grows. For example, a triple size of 635 for a package of size 2 consists of 140 tuples, whereas a package of size 5 has 175 tuples (a 25 % increase).

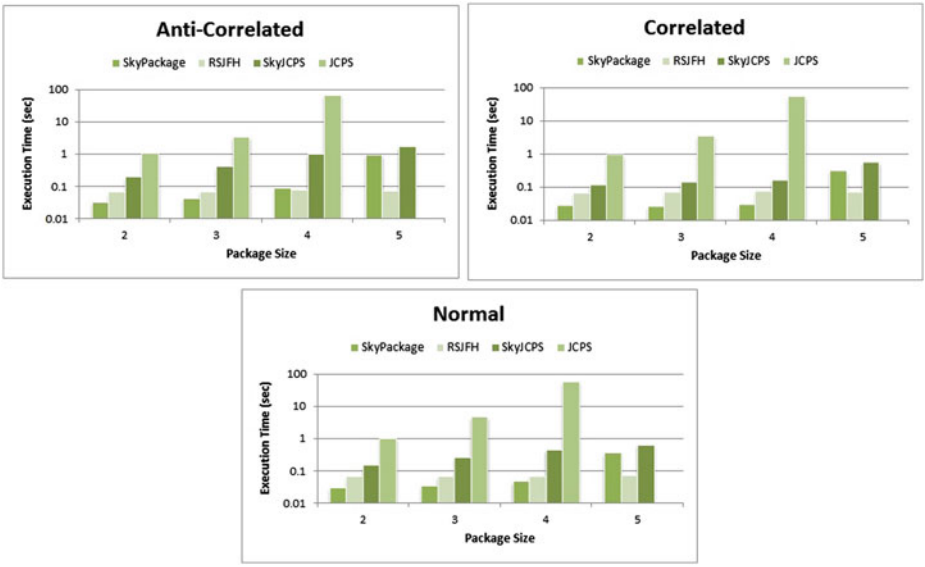


Fig. 16. Scalability for package sizes 2–5

From our experiment, we discovered that as the number of tuples increases, the percentage of skyline packages decreases. Also, while performing $SkyJCPS$, we found that the number of skyline tuples in the initial skyline phase increased as the number of tuples increased. Although the skyline size increased, the ratio between the skyline size and the number of tuples decreased, yielding a lower percentage. We argue that this is also the case with $SkyPackage$.

6.3 MovieLens Dataset

The first real-world dataset evaluated was MovieLens[2], which consisted of 10 million ratings and 10,000 movies.

Package-size Scalability and Prunability. We randomly chose a subset of users, with partiality to those who have rated a large number of movies, from the dataset for use in our package-size evaluations. The users consisted of those with IDs 8, 34, 36, 65, and 215, who rated 800, 639, 479, 569, and 1,242 movies, respectively. We used Query 1, where $n = 3, 4, 5$ for evaluation. The packages of size 3 consisted of users with IDs 8, 34, and 36, packages of size 4 included those three as well as the user with ID 65, and packages of size 5 included all five users. In formal notation, the query where $n = 3$ (a similar query is used for $n = 4, 5$) is $GP_{[r,X,\mathcal{F},PF]}$, where

[2] http://www.grouplens.org/node/73/

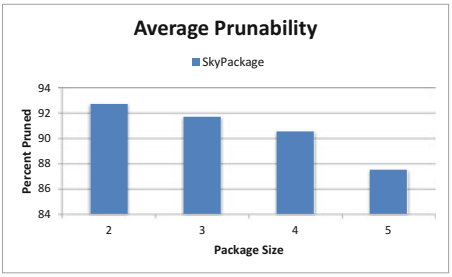


Fig. 17. Prunability of synthetic data

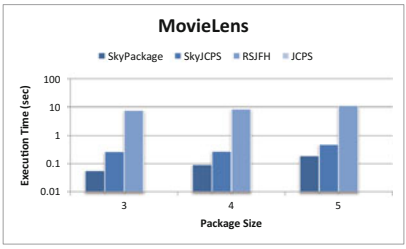


Fig. 18. Package size scalability for MovieLens

- $r = ?movie$
- $X = \{(?movieRater, hasName, \text{“8”}), (?movieRater, hasName, \text{“34”}), (?movieRater, hasName, \text{“36”})\ \}$
- $\mathcal{F} = \{SR = average()_{rating}\}$
- $PF = \{(SR; \prec_{max})\}$

The results of this experiment can be seen in Fig. 18. It is easily observed that *SkyPackage* performed better in all cases. We were unable to obtain any results from *JCPS* as it ran for hours. The next worst performing algorithm was *RSJFH*, followed by *SkyJCPS*. Due to the number of joins required to construct the tables in the format required by *RSJFH*, most of its time was spent during the initial phase, i.e., before the Cartesian product phase. Figure 19 shows the percent of tuples pruned by *SkyPackage*. In all three package sizes, approximately 99 % of the tuples were pruned.

6.4 Book-Crossing Dataset

The next real dataset used for evaluations was Book-Crossing[3], which consists of approximately 271,000 books rated by approximately 278,000 users.

Package-size Scalability and Prunability. Using a similar approach as we did with the MovieLens dataset, we randomly chose a subset of users for evaluating packages of different sizes. The users consisted of those with IDs 11601,

[3] http://www.informatik.uni-freiburg.de/~cziegler/BX/dataset

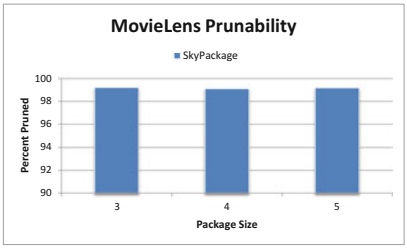


Fig. 19. Prunability of MovieLens dataset

11676, 16795, 23768, and 23902, who rated 1,571, 13,602, 2,948, 1,708, and 1,503 books, respectively. The target descriptive table contained approximately 271,000 tuples, i.e., all the books. We used Query 2, where $n = 3, 4, 5$ for evaluation.

Query 2. *Given n book-raters, find packages of n books such that the average rating of all the books is high and each book-rater has rated at least one of the books.*

In formal notation, the query where $n = 3$ (a similar query is used for $n = 4, 5$) is $GP_{[r,X,\mathcal{F},PF]}$, where

- $r = ?book$
- $X = \{(?bookRater, hasName, \text{“11601”}), (?bookRater, hasName, \text{“11676”}), (?bookRater, hasName, \text{“16795”})\ \}$
- $\mathcal{F} = \{SR = average()_{rating}\}$
- $PF = \{(SR; \prec_{max})\}$

The packages of size 3 consisted of users with IDs 11601, 11676, and 16795, packages of size 4 included those three as well as the user with ID 23768, and packages of size 5 included all five users. The results of this experiment can be seen in Fig. 20. The Book-Crossing dataset followed the same performance pattern as the MovieLens datasets, i.e., *SkyPackage* performed the best while *RSJFH* performed the worst. Although the overall execution time of the Book-Crossing dataset was longer than the MovieLens dataset, the data we used from the Book-Crossing dataset consisted of more tuples. Fig. 21 shows the percent of tuples pruned by *SkyPackage* from the Book-Crossing dataset.

6.5 Storage Model Evaluation

For each of the above experiments, we evaluated our storage model by comparing the time it took to load the RDF file into the database using our storage model versus using VPTs. All data was indexed using a B-trees.

For the synthetic data, we took the average time over the three data distributions of the same package size. Figure 22a shows the results for packages of

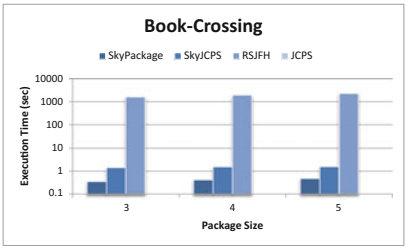


Fig. 20. Package size scalability for Book-Crossing

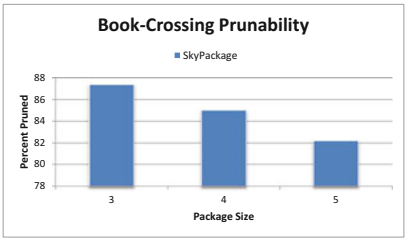
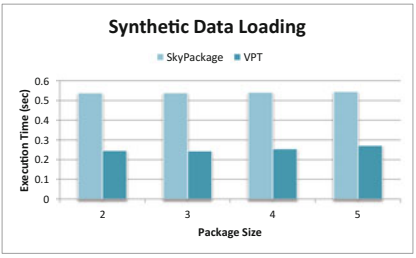
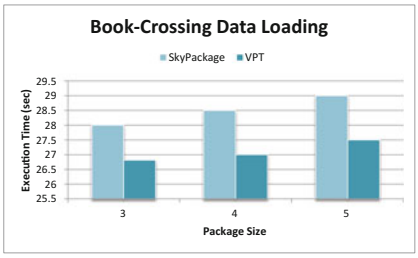


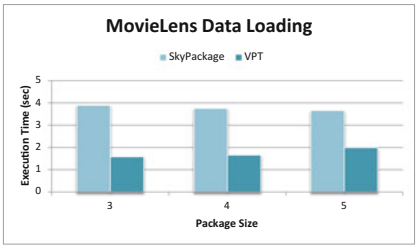
Fig. 21. Prunability of Book-Crossing dataset



(a) Synthetic Data



(b) Book-Crossing Data



(c) MovieLens Data

Fig. 22. Database build

size 2, 3, 4, and 5. In Fig. 22b, c, we show the time of inserting the MovieLens and the Book-Crossing datasets, respectively, for packages of size 3, 4, and 5.

For both the synthetic and MovieLens datasets, loading the data using our storage model took longer than using VPTs. The number of tables created using both approaches are not always equal, and either approach could have more tables than the other. Since the time to load the data is roughly the same for each package size, the number of tables created does not necessarily have that much effect on the total time. Our approach imposes additional time because of the triple patterns that must be matched, as explained in Sect. 4.1. Since the time difference between the two is small and the database only has to be built once, it is more efficient to use our storage model with *SkyPackage* than using VPTs.

7 Conclusion and Future Work

This article addressed the problem of answering package skyline queries. We have formalized and described what constitutes a "package" and have defined the term *skyline packages*. Package querying is especially useful for cases where a user requires multiple objects to satisfy certain constraints. We introduced three algorithms for solving the package skyline problem. Future work will consider the use of additional optimization techniques such as prefetching to achieve additional performance benefits as well as the integration of top-k techniques to provide ranking of the results when the size of query result is large.

Acknowledgment. The work presented in this article is partially funded by NSF grant IIS-0915865.

References

1. Abadi, D., Marcus, A., Madded, S., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB, Vienna (2007)
2. Barndorff-Nielsen, O., Sobel, M.: On the distribution of the number of admissable points in a vector random sample. Theor. Probab. Appl. **11**(2), 249–269 (1996)
3. Beckett, D.: RDFXML syntax specification. Recommendation, World Wide Web Consortium (2004). See http://www.w3.org/TR/rdf-syntax-grammar/
4. Beckett, D., Berners-Lee, T.: Turtle - Terse RDF triple language. World Wide Web Consortium (2011). See http://www.w3.org/TeamSubmission/turtle/
5. Bentley, J.L., Clarkson, K.L., Levine, D.B.: Fast linear expected-time algorithms for computing maxima and convex hulls. In: SODA, pp. 179–187 (1990)
6. Bentley, J.L., Kung, H.T., Schkolnick, M., Thompson, C.D.: On the average number of maxima in a set of vectors and applications, pp. 536–543. ACM (1978)
7. Berners-Lee, T., Fielding, R., Irvine, U.C., Masinter, L.: Uniform resource identifiers. (URI), Generic Syntax. IETF (1998)
8. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientif. Am. **284**(5), 34–43 (2001)

9. Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simeon, J.: XQuery 1.0: An XML Query Language, 2nd edn. Recommendation, World Wide Web Consortium (2010). See http://www.w3.org/TR/xquery/
10. Borzsonyi, S., Kossmann, D., Stocker, K.: The Skyline operator. In: ICDE, Heidelberg, pp. 421–430 (2001)
11. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0. Recommendation, World Wide Web Consortium (2000). See http://www.w3.org/TR/2008/REC-xml-20081126/
12. Brickley, D., Guha, R.V.: Resource Description Framework (RDF) Schema Specification 1.0 Canditate Recommendation, World Wide Web Consortium (2000). See http://www.w3.org/TR/2000/CR-rdf-schema-20000327
13. Broekstra, J.: SeRQL: Sesame RDF query language. SWAD-Europe, pp. 55–68 (2003)
14. Broekstra, J., Kampman, A., Harmelen, F.V.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In: ISWC, pp. 54–68 (2012)
15. Carrol, J., McBride, B.: The Jena semantic web toolkit. Public API, HP-Labs, Bristol (2001). See http://www.hpl.hp.com/semweb/jena-top.html
16. Chen, L., Gao, S., Anyanwu, K.: Efficiently evaluating skyline queries on RDF databases. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part II. LNCS, vol. 6644, pp. 123–138. Springer, Heidelberg (2011)
17. Chomicki, J.: Preference formulas in relational queries. TODS **28**(4), 427–466 (2003)
18. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: ICDE, pp. 717–719 (2003)
19. Grant, J., Beckett, D.: RDF test cases. Recommendation, World Wide Web Consortium (2004). See http://www.w3.org/TR/rdf-testcases/#ntriples
20. Deng, T., Fan, W., Geerts, F.: On the complexity of package recommendation problems. PODS, pp. 261–272 (2012)
21. Feigenbaum, L., Williams, G.T., Clark, K.G., Torress Hayes, E.: SPARQL 1.1 Protocol. Working draft. World Wide Web Consortium (2001). See http://www.w3.org/TR/sparql11-protocol/
22. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A declarative query language for RDF. World Wide Web, pp. 592–603 (2002)
23. Khabbaz, M., Lakshmanan, L.V.S.: TopRecs: Top-k algorithms for item-based collaborative filtering. EDBT, 213–224 (2011)
24. Khalefa, M.E., Mokbel, M.F., Levandoski, J.J.: Skyline query processing for incomplete data (2008)
25. KieBling, W.: Foundations of preferences in database systems. In: VLDB, pp. 311–322 (2002)
26. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: VLDB, pp. 275–286 (2002)
27. Koudas, N., Li, C., Tung, A.K.H., Vernica, R.: Relaxing join and selection queries. In: VLDB (2006)
28. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. JACM **22**(4), 469–476 (1975)
29. Lassila, O., Swick, R.R.: Resource description framework (RDF): Model and syntax specification. Recommendation, World Wide Web Consortium (1999). See http://www.w3.org/TR/REC-rdf-syntax/
30. Berners-Lee, T., Connolly, D.: Notation3 (N3): A readable RDF syntax. World Wide Web Consortium (2011). See http://www.w3.org/TeamSubmission/n3/

31. McLain, D.H.: Drawing contours from arbitrary data points. Comput. J. **17**(4), 318–324 (1974)
32. Raghavan, V., Rundensteiner, E.: SkyDB: Skyline aware query evaluation framework. In: IDAR (2009)
33. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. TODS **24**(2), 41–82 (2005)
34. Preparata, F., Shamos, M.: Computational Geometry: An Introduction. Springer, Heidelberg (1985)
35. Seaborne, A.: RDQL - A query language for RDF. World Wide Web Consortium (2004). See http://www.w3.org/Submission/RDQL/
36. Sessoms, M., Anyanwu, K.: SkyPackage: From finding items to finding a skyline of packages on the semantic web. Proceedings of the JIST (to appear), (2012)
37. Shah, K., Gadge, J.: Semantic web services for E learning: Engineering and technology domain. In: IJCTE 2001, pp. 727–731 (2001)
38. 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)
39. Sintek, M., Decker, S.: TRIPLE–A query, inference, and transformation language for the semantic web. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, p. 364. Springer, Heidelberg (2002)
40. Souzis, A.: RxPath specification proposal. See http://rx4rdf.liminalzone.org/RxPathSpec
41. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: VLDB, pp. 301–310 (2001)
42. Uschold, M., Gruninger, M.: Ontologies: principles, methods and applications. Knowl. Eng. Rev. **11**(2), 93–136 (1996)
43. Vlachou, A., Doulkeridis, C., Polyzotis,N.: Skyline query processing over joins. In: SIGMOD, pp. 73–84 (2011)
44. Xie, M. Lakshmanan, L.V.S., Wood, P.T.: Breaking out of the box of recommendations: from items to packages. In: RecSys, pp. 151–158 (2010)
45. Xie, M. Lakshmanan, L.V.S., Wood, P.T.: CompRec-Trip: a composite recommendation system for travel planning. In: ICDE, pp. 1352–1355 (2011)
46. Yiu, M.L., Mamoulis, N.: Efficient processing of top-k dominating queries on multi-dimensional data. In: VLDB, pp. 483–494 (2007)
47. Jin, W., Ester, M., Hu, Z., Han, J.: The Multi-relational skyline operator. In: ICDE, pp. 1276–1280 (2007)
48. Guo, X., Xiao, C., Ishikawa, Y.: Combination skyline queries. In: Hameurlain, A., Küng, J., Wagner, R., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) Transactions on Large-Scale Data- and Knowledge-Centered Systems VI. LNCS, vol. 7600, pp. 1–30. Springer, Heidelberg (2012)