SkyPackage: From Finding Items to Finding a Skyline of Packages on the Semantic Web

Matthew Sessoms and Kemafor Anyanwu

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

Abstract. Enabling complex querying paradigms over the wealth of available Semantic Web data will significantly impact the relevance and adoption of Semantic Web technologies in a broad range of domains. While the current predominant paradigm is to retrieve a list of items, in many cases the actual intent is satisfied by reviewing the lists and assembling compatible items into lists or packages of resources such that each package collectively satisfies the need, such as assembling different collections of places to visit during a vacation. Users may place constraints on individual items, and the compatibility of items within a package is based on global constraints placed on packages, like total distance or time to travel between locations in a package. Finding such packages using the traditional item-querying model requires users to review lists of possible multiple queries and assemble and compare packages manually.

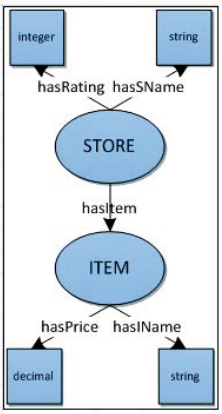
In this paper, we propose three algorithms for supporting such a package query model as a first class paradigm. Since package constraints may involve multiple criteria, several competing packages are possible. Therefore, we propose the idea of computing a skyline of package results as an extension to a popular query model for multi-criteria decision-making called skyline queries, which to date has only focused on computing item skylines. We formalize the semantics of the logical query operator, *SkyPackage*, and propose three algorithms for the physical operator implementation. A comparative evaluation of the algorithms over real world and synthetic-benchmark RDF datasets is provided.

Keywords: RDF, Package Skyline Queries, Performance.

1 Introduction

The Linking Open Data movement and broadening adoption of Semantic Web technologies has created a surge in amount of available data on the Semantic Web. The rich structure and semantics associated with such data provides an opportunity to enable more advanced querying paradigms that are useful for different kinds of tasks. An important class of such advanced querying models is the *skyline query* model for supporting multi-criteria retrieval tasks. While

H. Takeda et al. (Eds.): JIST 2012, LNCS 7774, pp. 49–64, 2013.



&S1	hasSName	a
&S2	hasSName	b
&S3	hasSName	g
&S4	hasSName	e

&S1	hasRating	5
&S2	hasRating	5
&S3	hasRating	18
&S4	hasRating	22

&S1	hasItem	&I1
&S1	hasItem	&I2
&S1	hasItem	&I3
&S2	hasItem	&I4
&S2	hasItem	&I5
&S2	hasItem	&I6
&S3	hasItem	&I7
&S3	hasItem	&I8
&S3	hasItem	&I9
&S4	hasItem	&I10
&S4	hasItem	&I11
&S4	hasItem	&I12

&I1	hasIName	milk
&I2	hasIName	eggs
&I3	hasIName	bread
&I4	hasIName	milk
&I5	hasIName	eggs
&I6	hasIName	bread
&I7	hasIName	milk
&I8	hasIName	eggs
&I9	hasIName	bread
&10	hasIName	milk
&I11	hasIName	eggs
&I12	hasIName	bread

&I1	hasPrice	2
&I2	hasPrice	7
&I3	hasPrice	5
&I4	hasPrice	3
&I5	hasPrice	10
&I6	hasPrice	4
&I7	hasPrice	8
&I8	hasPrice	8
&I9	hasPrice	8
&I10	hasPrice	9
&I11	hasPrice	8
&I12	hasPrice	3

	b	g	e		
Price	3	8	3	Total:	$14
Rating	5	18	22	Average:	15

	a	e	e		
Price	2	8	3	Total:	$13
Rating	5	22	22	Average:	16.33

	b	e	e		
Price	3	8	3	Total:	$14
Rating	5	22	22	Average:	16.33

Fig. 1. Data For E-Commerce Example

this class of queries have been considered as an extension to traditional pattern matching queries in relational [2][7][10] and more recently Semantic Web data models [3], it is important to consider more complex underlying query models. While pattern matching paradigms are predicated on finding individual single items, often times a user is interested in finding item combinations or packages that satisfy certain constraints. For example, a student may be interested in finding a combination of e-learning resources that cover a set of topics, with constraints like overall rating of resources should be high, and if being purchased, total cost should be low.

As a more concrete example, assume stores expose Semantic Web enabled catalogs, and a customer wishes to make a decision about what combination of stores would best meet his or her purchase needs. The customer wishes to purchase milk, eggs, and bread and is willing to make the purchase from multiple stores as long as the total price spent is minimized, and the overall average rating across stores is good or maximized, i.e., the stores are known to have good quality products. Additional constraints could include minimizing total distance traveled between the stores and some constraints about open shopping hours.

The example shows that the target of the query is in fact a set of items (a set of stores), where the query should return multiple combinations or "packages". Since the preferences are specified over aggregated values for elements in a package, the process of producing combinations will need to precede the preference checking. One possible package for the e-commerce example, given the sample data in Fig 1, 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.

In [5][11][12], the idea of package recommendation has been explored. This problem assumes a single criterion such that an optimal package can be selected or a top-k of packages based on techniques similar to collaborative filtering for recommendation. With respect to skyline queries, different indexing strategies for optimizing skyline computation over a single relation or restricted join structures e.g. just two relations i.e. single join [10], star-like schemas [14] have been investigated. The challenges of skylining over RDF data models which do not conform to the requirement of being contained in two relations or having restricted schema structures, were explored in [3]. The proposed approach interleaved the join and skyline phases in such a way that the information about the already explored values was used to prune the non-valuable candidates from entering the skyline phase. However, these existing techniques all focus on computing item-based skylines. The main challenge in terms of package skylines is dealing with the increase in search space due to the combinatorial explosion.

In this paper, we introduce the concept of *package skyline queries* and propose query evaluation techniques for such queries. 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 three algorithms proposed over synthetic and real world data. The rest of the paper is organized as follows. The background and formalization of our problem is given in Section 2. Section 3 introduces the two algorithms based on the vertical partitioned storage model, and Section 4 presents an algorithm based on our TDTQ storage model. An empirical evaluation study is reported in Section 5, and related work is described in Section 6. The paper is concluded in Section 7.

2 Preliminaries

Specifying the e-commerce example as a query requires the use of a graph pattern structure for describing the *target* of the query (*stores*), and the *qualification* for the desired targets e.g. stores *should sell* at least one of *milk* or *bread* or *eggs*. We call these *target qualifiers*. The second component of the query specifications concerns the *preferences*, e.g., *minimizing the total cost*. In our example, preferences are specified over the aggregates of target attributes (datatype properties), e.g., maximizing the average over store ratings, as well as the attributes of target qualifiers, e.g., minimizing total price. Note that the resulting graph pattern structure (we ignore the details of preferences specification at this time) involves a union query where each branch computes a set of results for one of the target qualifiers, shown in Fig 2.

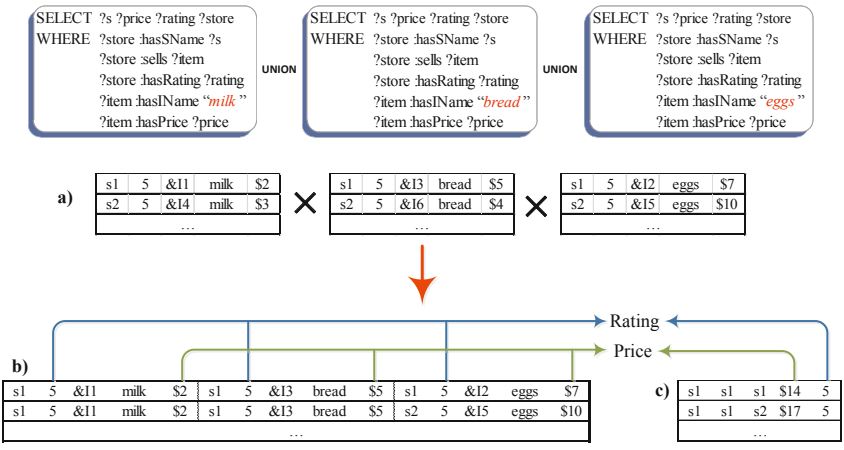


Fig. 2. Dataflow for the Skyline Package Problem in Terms of Traditional Query Operators

2.1 Problem Definition

Let D be a dataset with property relations P_1, P_2, .., P_m and GP be a graph pattern with triple patterns TP_i, TP_j, .., TP_k (TP_x denotes triple pattern with property P_x). $[[\text{GP}]]_D$ denotes the answer relation for GP over D, i.e., $[[\text{GP}]]_D = P_i \bowtie P_j \bowtie ... \bowtie P_k$. Let $var(TP_x)$ and $var(GP)$ denote the variables in the triple and graph pattern respectively. We designate the return variable $r \in var(GP)$, e.g., (*?store*), representing the target of the query (stores) as the *target* variable. We call the triple patterns such as (*?item hasName "milk"*) *target qualifying constraints* since those items determine the stores that are selected.

We review the formalization for preferences given in [7]. Let $Dom(a_1)$, ..., $Dom(a_d)$ be the domains for the columns $a_1, .., a_d$ in a d-dimensional tuple $t \in [[\text{GP}]]_D$. Given a set of attributes B, then a preference PF over the set of tuples $[[\text{GP}]]_D$ is defined as $PF := (B; \succ_{PF})$ where $\succ_{PF}$ is a strict partial order on the domain of B. Given a set of preferences $PF_1, ..., PF_m$, their combined Pareto preference PF is defined as a set of equally important preferences.

For a set of d-dimensional tuples R and preference $PF = (B; \succ_{PF})$ over R, a tuple $r_i \in R$ dominates tuple $r_j \in R$ based on the preference (denoted as $r_i \succ_{PF} r_j$), iff $(\forall(a_k \in B)(r_i[a_k] \succeq r_j[a_k]) \wedge \exists(a_l \in B)(r_i[a_l] \succ r_j[a_l]))$

Definition 1 (Skyline Query). *When adapting preferences to graph patterns we associate a preference with the property (assumed to be a datatype property) on whose object the preference is defined. Let PF_i denote a preference on the column representing the object of property P_i). Then, for a graph pattern $GP = TP_1, .., TP_m$ and a set of preferences $PF = PF_i, PF_j, .., PF_k$, 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 (Map and Generalized Projection). . *Let $\mathcal{F} = \{f_1, f_2, ...f_k\}$ 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 [7]) 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 $\mathcal{X} = \{x_1, x_2, ...x_k\}$ 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 $R'(colr_x, colr_y, colr_z, .., x_1, x_2, ...x_k)$. 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 Query). . *A* SkyPackage *graph pattern query is graph pattern $GP_{[\{c_1,c_2,...,c_N\},\mathcal{F}=\{f_1,f_2,...,f_k\}\{PF_{P_i},PF_{P_j},...PF_{P_k}\},r]}$ such that :*

1. *c_i is a qualifying constraint.*
2. *$r \in \text{var}(GP)$ is called the target of the query e.g., stores.*
3. *PF_{P_i} is the preference specified on the property P_i, i.e., actually the object of p_i. f_i is the mapping function of P_i.*

Definition 4 (SkyPackage Query Answer). *The* answer *to a* skypackage *graph pattern query R_{SKY} can be seen the result of the following steps:*

1. *$R_{product} = [[GP_{c_1}]] \times [[GP_{c_2}]] \times ... \times [[GP_{c_N}]]$ such that $[[GP_{c_x}]]$ is the result of evaluating the branch of the union query with constraint c_x.* Fig 2.(b) shows the partial result of the crossproduct of the three subqueries in (a) based on the 3 constraints on milk, bread and eggs.
2. *$R_{project} = \prod_{r_1,r_2,...r_N,\hat{\mu}_{[\mathcal{F}=\{f_1,f_2,...f_k\},\mathcal{X}=\{x_1,x_2,...x_k\}]}}(R_{product})$ where r_i is the column for the return variable in subquery i's result, $f_1 : (domc1(o_1) \times domc2(o_1) \times ... \times domcN(o_1)) \rightarrow \mathbb{R}$ where $domc1(o_1)$ 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 2.(c)
3. *$SKYLINE[R_{project}, \{PF_{P'_i}, PF_{P'_j}, ...PF_{P'_k}\}]$ 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 totalprice.*

In other words, the result of a skypackage query is the result of skylining over the extended generalized projection (with map functions consisting of desired aggregates) of the combination (product) relation.

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:

1. $I \leftarrow hasIName \bowtie hasSName \bowtie hasItem \bowtie hasPrice \bowtie hasRating$
2. Perform Cartesian product on I twice (e.g., $I \times I \times I$)
3. Aggregate *price* and *rating* attributes
4. Perform a single-table skyline algorithm

Steps 1 and 2 can be seen in Fig 3a, which depicts the Cartesian product being performed twice on I to obtain all store packages of size 3. 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.

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

✖

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

✖

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 $JCPS$

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 Cartesian products are performed on I (lines 3-5). Afterwards,

equivalent skyline attributes are aggregated (lines 6-8). 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 9 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 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 $\mathcal{P}$

```
1: n ← package size
2: I ← VPT_1 ⋈ VPT_2 ⋈ ··· ⋈ VPT_x
3: for all i ∈ [1, n − 1] do
4:     I ← I × I
5: end for
6: for all i ∈ [1, y] do
7:     I ← A_{s_i}(I)
8: end for
9: P ← skyline(I)
10: return P
```

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 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. 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 $\rho = \{p_1 p_2 \ldots p_n\}$ and $\rho' = \{p_1 p_2 \ldots p'_n\}$ be packages of size n and ρ be a skyline package. If $p_n \preceq_{C_n} p'_n$, where C_n is a qualifying constraint, then ρ' is not a skyline package.*

Proof. Let $x_1, x_2, \ldots, x_m$ 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 an 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, $RSJFH$ [3], 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

Fig 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

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 $\mathcal{P}$

1: $n \leftarrow$ package size
2: $t_1, t_2, \ldots, t_n \leftarrow$ targets of the package
3: VPT_1 contains targets and VPT_2 contains a skyline attribute of the targets
4: $I^2 \leftarrow VPT_3 \bowtie \cdots \bowtie VPT_x$
5: **for all** $i \in [1, n]$ **do**
6: $\quad I_i^1 \leftarrow \sigma_{t_i}(VPT_1) \bowtie VPT_2$
7: $\quad S_i \leftarrow RSJFH(I_i^1, I^2)$
8: **end for**
9: $T \leftarrow S_1 \times S_2 \times \cdots \times S_n$
10: **for all** $i \in [1, n]$ **do**
11: $\quad I \leftarrow \mathcal{A}_{s_y}(T)$
12: **end for**
13: $\mathcal{P} \leftarrow$ skyline(I)
14: **return** $\mathcal{P}$

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 $JCPS$.

4 Algorithms for Package Skyline Queries over the TDTQ Storage Model

In this section, we present a more efficient and feasible method to solve the skyline package problem. We discuss a devised storage model, *Target Descriptive, Target Qualifying* (TDTQ), and an overview of an algorithm, $SkyJCPS$, that exploits this 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.

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

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 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: $\mathcal{P} \leftarrow \text{skyline}(I)$
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. Fig 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) $I'_{TQ_i} \leftarrow \text{skyline}(I_{TQ_i})$
2. $CPJS(I'_{TQ_1}, \ldots, I'_{TQ_i}, rating)$

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.

5 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.

5.1 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 three algorithms, $JCPS$, $SkyJCPS$, and $RSJFH-CPS$. During the skyline phase of each of these algorithms, we used the *block-nested-loops* (BNL) [2] 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.

5.2 Package-Size Scalability - Synthetic Data

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.

In Fig 6 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 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.

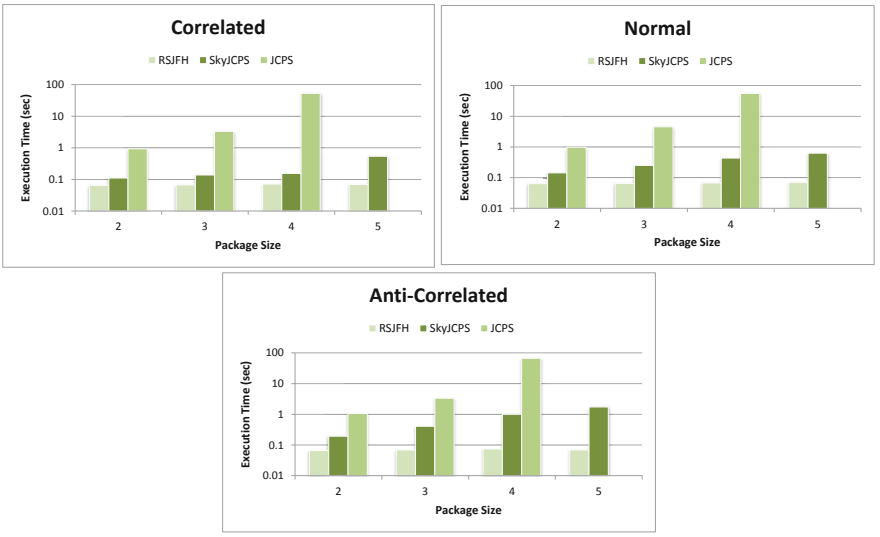


Fig. 6. Package Size Scalability for Synthetic Data

5.3 Package-Size Scalability - Real Dataset

In order to test the algorithms' performance with real data, we used the MovieLens[1] dataset, which consists of 10 million ratings and 10,000 movies, and the Book-Crossing[2], which consists of 271,000 books rated by 278,000 users.

We randomly chose five users from each of the datasets, with partiality to those who have rated a large number of movies or books, from the datasets for use in our package-size evaluations. For the MovieLens dataset, the users consisted of those with IDs 8, 34, 36, 65, and 215, who rated 800, 639, 479, 569, and 1,242 movies, respectively. Similarly, for the Book-Crossing dataset, the users consisted of those with IDs 11601, 11676, 16795, 23768, and 23902, who rated 1,571, 13,602, 2,948, 1,708, and 1,503 books, respectively. The target

[1] `http://www.grouplens.org/node/73/`
[2] `http://www.informatik.uni-freiburg.de/~cziegler/BX/dataset`

descriptive table for MovieLens contained 10,000 tuples, i.e., all the movies, and Book-Crossing contained 271,000 tuples, i.e., all the books. We used the following queries, respectively, for evaluating MovieLens and Book-Crossing datasets: *find packages of n movies such that the average rating of all the movies is high, the release date is high, and each movie-rater has rated at least one of the movie* and *find packages of n books such that the average rating of all the books is high, the publication date is high, and each book-rater has rated at least one of the books*, where $n = 3, 4, 5$.

The results of this experiment can be seen in Fig 10. It is easily observed that $SkyJCPS$ performed better in all cases. We were unable to obtain any results from $JCPS$ as it ran for hours. 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, and performed the worst. On average, $SkyJCPS$ outperformed $RSJFH$ by a factor of 1000. 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.

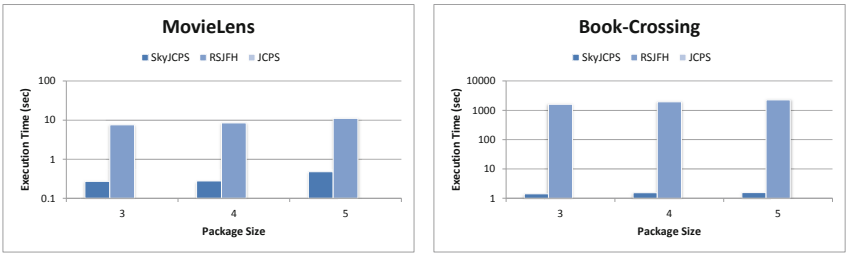


Fig. 7. Package Size Scalability for MovieLens and Book-Crossing Datasets

5.4 Storage Model Evaluation

For each of the above experiments, we evaluated our storage model by comparing the time taken to load the RDF file into the database using our storage model versus using VPTs. All data was indexed using a B-trees. Fig 8 shows the time of inserting the MovieLens and the Book-Crossing datasets, respectively, for packages of size 3, 4, and 5.

In general, the data loading phase using the TDTQ storage model was longer than that of 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. 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 $SkyJCPS$ than using VPTs.

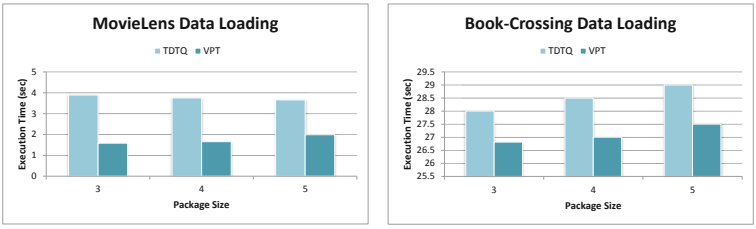


Fig. 8. Database build

6 Related Work

Much research has been devoted to the subject of preference queries. One kind of preference query is the skyline query, which [2] originally introduced and provided a block nested loops, divide-and-conquer, and B-tree-based algorithms. Later, [4] introduced a sort-filter-skyline algorithm that is based on the same intuition as BNL, but uses a monotone sorting function to all for early termination. Unlike [2] [4] [9], which have to read the whole database at least once, index-based algorithms [6] [8] allow one to access only a part. However, all of these are designed to work on a single relation. As the Semantic Web matures and RDF data is populated, more research needs to be done on preference queries that involve multiple relations. When queries involve aggregations, multiple relations must be joined before any of the above techniques can be used.

Recently, an interest in multi-relation skyline queries has been growing. [3] introduced three skyline algorithms that are based on the concept of a *header point*, which allows some nonskyline tuples to be discarded before succeeding to the skyline processing phase. [7] introduced a sky-join operator that gives the join phase a small knowledge of a skyline. Others have used approximation and top-k algorithms with regards to recommendation. [5] proposes a framework for collaborative filtering using a variation of top-k. However, their set of results do not contain packages but single items. [12] went a step further and used top-k techniques to provide a *composite* recommendation for travel planning. Also, [11] used similar techniques to provide a package recommendation by approximating. 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. [13] explains the concept of top-k dominating queries. They combine the properties of skyline and top-k to yield an algorithm that has the best of both worlds. [10] proposed a novel algorithm called SFSJ (sort-first skyline-join) that computes the complete skyline. Given two relations, access to the two relations is alternated using a pulling strategy, known as adaptive pulling, that prioritizes each relation based on the number of mismatched join values. Although the algorithm has no limitations on the number of skyline attributes, it is limited by two relations.

7 Conclusion and Future Work

This paper 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.

Acknowledgments. The work presented in this paper is partially funded by NSF grant IIS-0915865. Thanks to Ling Chen for the meaningful discussions and suggestions. Thanks to Alicia Bieringer, Juliane Foster, and Prachi Nandgaonkar for their fruitful discussions and participation in the initial stage of this work.

References

1. Abadi, D., Marcus, A., Madded, S., Hollenbach, K.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB 2007 (2007)
2. Borzsonyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE 2001, pp. 421–430 (2001)
3. 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)
4. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: ICDE 2003, pp. 717–719 (2003)
5. Khabbaz, M., Lakshmanan, L.V.S.: TopRecs: Top-k algorithms for item-based collaborative filtering. In: EDBT 2011, pp. 213–224 (2011)
6. Kossmann, D., Ramsak, F., Rost, S.: Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In: VLDB 2002, pp. 275–286 (2002)
7. Raghavan, V., Rundensteiner, E.: SkyDB: Skyline Aware Query Evaluation Framework. In: IDAR 2009 (2009)
8. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive Skyline Computation in Database Systems. In: TODS 2005, pp. 41–82 (2005)
9. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient Progressive Skyline Computation. In: VLDB 2001, pp. 301–310 (2001)
10. Vlachou, A., Doulkeridis, C., Polyzotis, N.: Skyline query processing over joins. In: SIGMOD 2011, pp. 73–84 (2011)
11. Xie, M., Lakshmanan, L.V.S., Wood, P.T.: Breaking out of the Box of Recommendations: From Items to Packages. In: RecSys 2010, pp. 151–158 (2010)
12. Xie, M., Lakshmanan, L.V.S., Wood, P.T.: CompRec-Trip: a Composite Recommendation System for Travel Planning. In: ICDE 2011, pp. 1352–1355 (2011)
13. Yiu, M.L., Mamoulis, N.: Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data. In: VLDB 2007, pp. 483–494 (2007)
14. Jin, W., Ester, M., Hu, Z., Han, J.: The Multi-Relational Skyline Operator. In: ICDE 2007, pp. 1276–1280 (2007)