

ABSTRACT

SESSOMS, MATTHEW WADE. SkyPackage: From Finding Items to Finding A Skyline of Packages on the Semantic Web. (Under the direction of Dr. Kemafor Anyanwu.)

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 and time to travel between locations in a package and total cost. 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 thesis, we propose four 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 problem and discuss our implementation strategy for a loose integration with an open source RDF system, and evaluate the algorithm using synthetic and real world datasets.

© Copyright 2012 by Matthew Wade Sessoms

All Rights Reserved

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

by
Matthew Wade Sessoms

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2012

APPROVED BY:

Dr. Christopher Healey

Dr. Nagiza Samatova

Dr. Kemafor Anyanwu
Chair of Advisory Committee

DEDICATION

To Mom, Dad, Jamie, and Samantha.

BIOGRAPHY

Matthew Wade Sessoms was born on October 13, 1986. He received a Bachelor of Science in Mathematics and Computer Science from Methodist University, Fayetteville, NC in 2010. He is currently enrolled as a master's student in Computer Science at North Carolina State University. He plans to continue his career as a software engineer at IBM, Research Triangle Park, NC.

ACKNOWLEDGEMENTS

First and foremost, I want to thank God for making all things possible. I want to thank my advisor, Dr. Kemafor Anyanwu, for her advice, efforts, and patience in guiding my research. I would like to thank my committee members, Dr. Christopher Healey and Dr. Nagiza Samatova, who spent time and efforts on reviewing my thesis. I am indebted to NC State for giving me the opportunity to gain invaluable experience and knowledge.

TABLE OF CONTENTS

List of Tables	vii
List of Figures	viii
Chapter 1 Introduction	1
1.1 Thesis Motivation	1
1.2 Key Contributions	2
1.3 Thesis Organization	3
Chapter 2 Background and Related Work	4
2.1 Preference Queries	4
2.1.1 Skyline Queries	5
2.2 Semantic Web	6
2.2.1 RDF Data Model	7
2.2.2 SPARQL	7
2.3 Motivating Example	9
2.4 Problem Definition	11
2.5 Related Work	13
2.5.1 Historical Perspective	14
2.5.2 Single-relation Skyline Algorithms	14
2.5.3 Multi-relation Skyline Algorithms	14
2.5.4 Composite Top- k Algorithms	15
Chapter 3 Techniques for Evaluating Skyline Packages	16
3.1 Algorithms for Package Skyline Queries over Vertical Partitioned Tables	16
3.1.1 <i>JCPS</i> Algorithm	16
3.1.2 <i>RSJFH – CPS</i> Algorithm	18
3.2 Algorithms for Package Skyline Queries over the TDTQ Storage Model	21
3.2.1 The TDTQ Storage Model	22
3.2.2 Notations	23
3.2.3 SkyPackage Algorithm	25
Chapter 4 Sesame Integration Framework	31
4.1 Sesame	31
4.2 Framework	32
4.2.1 Data Storage	33
4.2.2 Data Retrieval	35

Chapter 5 Evaluation	37
5.1 Overview	37
5.2 Synthetic Data	38
5.2.1 Dataset	38
5.2.2 Data Size Scalability	38
5.2.3 Package Size Scalability	42
5.2.4 Average Prunability	42
5.3 MovieLens Dataset	44
5.3.1 Package-size Scalability and Prunability	45
5.4 Book-Crossing Dataset	46
5.4.1 Package-size Scalability and Prunability	46
5.5 Storage Model Evaluation	47
Chapter 6 Conclusion and Future Work	51
6.1 Conclusion	51
6.2 Future Work	52
References	53

LIST OF TABLES

Table 2.1	Triple notation for Figure 2.2	7
-----------	--	---

LIST OF FIGURES

Figure 2.1	Skyline of Hotels	5
Figure 2.2	RDF Graph	8
Figure 2.3	Data For E-Commerce Example	10
Figure 2.4	Dataflow for the SkyPackage Problem in Terms of Traditional Query Operators	11
Figure 3.1	Cartesian Product of Join Result	17
Figure 3.2	Cartesian Product Result	17
Figure 3.3	<i>RSJFH</i> (skyline-over-join) for milk	20
Figure 3.4	<i>RSJFH</i> 's result for milk	20
Figure 3.5	Cartesian product on all targets (e.g., milk, eggs, and bread)	20
Figure 3.6	E-commerce Data	22
Figure 3.7	Skyline Region	26
Figure 3.8	Pruning Example	27
Figure 3.9	Pruning and Early Termination Result	29
Figure 3.10	Skyline Package Result	29
Figure 4.1	Sesame Components	32
Figure 4.2	MovieLens Ontology	33
Figure 4.3	Framework with SPARQL Adapter	34
Figure 4.4	Queries to construct target qualifying tables	36
Figure 4.5	Query to construct target descriptive table	36
Figure 5.1	Synthetic Data Triple Sizes	39
Figure 5.2	Evaluation Results for Package Sizes 2 and 3	40
Figure 5.3	Evaluation Results for Package Sizes 4 and 5	41
Figure 5.4	Scalability for Package Sizes 2 to 5	43
Figure 5.5	Prunability of Synthetic Data	44
Figure 5.6	Package Size Scalability for MovieLens	45
Figure 5.7	Prunability of MovieLens Dataset	46
Figure 5.8	Package Size Scalability for Book-Crossing	47
Figure 5.9	Prunability of Book-Crossing Dataset	48
Figure 5.10	Database build for Synthetic Data	49
Figure 5.11	Database build for MovieLens Data	49
Figure 5.12	Database build for Book-Crossing Data	50

Chapter 1

Introduction

The traditional search and querying paradigms on the Web are aimed towards a single item being the focus of a user's item search. With the Semantic Web envisioning documents from different origins being linked together, the data becomes more useful enabling data from different sources to be connected and queried. The surge in interest and availability of structured data on the Semantic Web provides an opportunity to investigate more advanced querying paradigms that are useful for different kinds of tasks, specifically finding a combination or package of items satisfying a user's preference.

Skyline queries, a common type of preference queries, focuses on finding the best set of objects, with respect to the user's preferences, from the given data. Current research focuses primarily on skyline sets whose elements have cardinality of one. The Semantic Web allows for more advanced searches involving not only single-item sets, but also multi-item sets (i.e., packages). The main challenge in terms of package skylines is dealing with the exponential increase in the search space due to the combinatorial explosion.

1.1 Thesis Motivation

Consider a customer that 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)

meaning the goal of the query is to find candidate combinations or “packages.” Finding such packages using the traditional item querying model requires users to review lists of results for possibly multiple queries and then manually combining these into suitable packages which can be very tedious.

To illustrate another example, consider Samantha, who is in the 8th grade, is having difficulty grasping the concepts of mathematics. She wishes to pursue additional help by means of e-learning and is interested in videos that cover quadratic equations, linear functions, and scatterplots. Each video, i.e., a resource, has properties such as rating and video length. She is willing to spend up to 45 minutes each day using an e-learning system and prefers the average rating of all the videos to be high. Hence, Samantha is interested in a package of three e-learning resources whose total property value for length is less than 45 and whose average property value for rating is high.

These examples show that the target of the query is in fact a set of items (a set of stores, a set of learning sessions), where the query should return multiple combinations or “package”. Finding such packages using the traditional item querying model requires users to review lists of results for possible multiple queries and then manually combining these into suitable packages. The examples also demonstrate the need to support multiple selection criteria for packages. Furthermore, selection criteria could involve hard constraints, such as a total price for the package is less than \$500, or soft constraints, such as preferring a minimally priced item.

1.2 Key Contributions

To our knowledge, we are the first to study package skyline queries. In this thesis, we propose the concept of package skyline queries and an approach for evaluating such queries. Specifically, we contribute the following:

1. A formalization of the concept of package skyline queries over an RDF data model
2. An efficient storage model for making such queries easier to answer.
3. An efficient technique for evaluating package skyline queries on an RDF database
4. An approach for a loose integration of this technique in an open source RDF database, Sesame

1.3 Thesis Organization

The remaining thesis is organized as follows:

Chapter 2 provides the necessary background information, such as skyline queries, RDF, and SPARQL, to understand our proposed approach. Also, the motivating example is further examined, the problem is formally stated, and related work is provided. Chapter 3 provides techniques for answering package skyline queries and introduces our proposed storage model and algorithm. Chapter 4 presents our implementation with which we loosely integrate our approach into an RDF database, called Sesame. Chapter 5 illustrates the effectiveness and efficiency of our approach. An outline of our experiment and a presentation of our results is given. Chapter 6 presents the conclusion and suggests direction for future work.

Chapter 2

Background and Related Work

This chapter introduces preference queries and a type of preference query known as skyline queries. It also provides some definitions and concepts that will be used throughout this thesis. Finally, we provide related work.

2.1 Preference Queries

Preferences, such as “I like A more than B,” can be easily stated by users when asked for their wishes. For example, when purchasing a car, it is easy to state a preferred color and that a car with high gas mileage is more desirable than one with lower gas mileage. Preferences consists of *soft constraints* and *hard constraints*. A hard constraint represents an absolute limitation on the choices at hand, while a soft constraint is neither required nor necessary, but desired. If a preference query contained only hard constraints, it is likely that the search results would not yield many, if any, exact matches, forcing a low number of results, and thus soft constraints are more suitable for users.

Preference-based querying [17][25][36] involves presenting users with a set of *best* possible answers that best satisfy the query even if no exact matches are found. A common type of preference query is a *skyline query*, which identifies the best matches according to the given preferences.

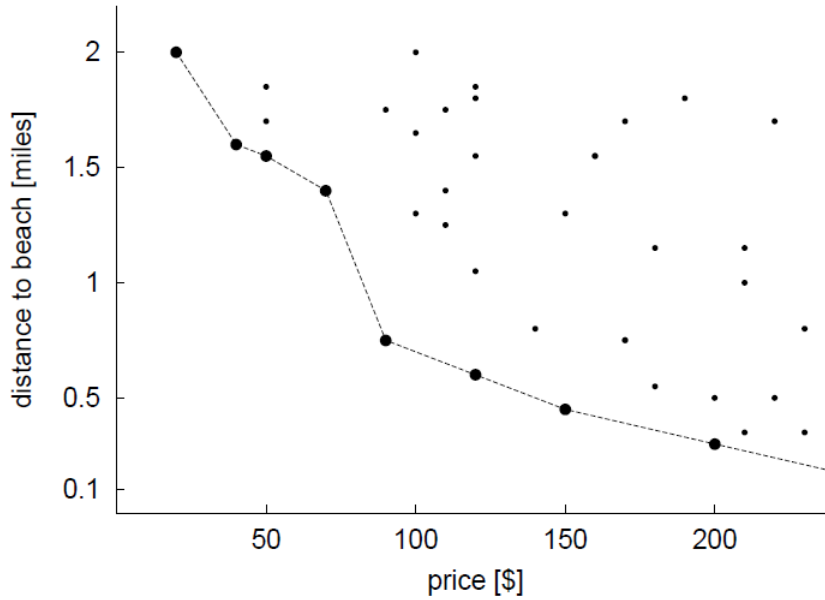


Figure 2.1: Skyline of Hotels

2.1.1 Skyline Queries

A classic example used to illustrate the concept of skyline queries is searching for cheap hotels close to the beach. Suppose John and his family are visiting the beach for a weekend vacation and are interested in finding a hotel that is close to the beach and has a low price. If hotel h_1 is closer to the beach and has a cheaper price than hotel h_2 , it is obvious that h_1 is a better choice than h_2 . In this case, h_1 is said to *dominate* h_2 in both price and distance. If h_1 is cheaper than h_2 but is located farther away from the beach, h_1 and h_2 are called *incomparable* (neither is better assuming equal weight on preferences).

A sample set of hotels is depicted in Figure 2.1 (borrowed from [10]), characterized by two dimensions, distance and price. A hotel belongs in the skyline if there are no other hotels that are better than it in both dimensions, i.e., both cheaper and closer to the beach. Similarly, a hotel appears in the skyline if, at a given distance from the beach, no hotel is cheaper, or conversely if, at a given price, no hotel is closer to the beach. The distinguishing property of the skyline is that for any preference function f that is monotone on all attributes, if an object maximizes f , then this object is part of the skyline. Also, for every object in the skyline, there exists a monotone preference

function that is maximized by this object. Intuitively, this means that (a) regardless of how a user weighs his/her preferences, his/her top preferred object will be one of the skyline objects, and (b) there is no skyline object which is nobody's top preference. The skyline is formally defined next.

Definition 2.1.1 (Dominance) *Let $p = (p_1, \dots, p_d)$ and $p' = (p'_1, \dots, p'_d)$ be two points in \mathcal{R}^d . Point p dominates p' (denoted as $p \preceq p'$) if and only if p_i is better than or equal to p'_i for $1 \leq i \leq d$ and p_j is strictly better than p'_j for some $1 \leq j \leq d$. The total order \preceq is reflexive, antisymmetric, transitive, and comparable (totality).*

In other words, given a set of points, a point dominates another point if it is as good or better in all dimensions and better in at least one dimension.

Definition 2.1.2 (Skyline Set) *The set of all non-dominated points.*

The problem of finding all such points is called the *skyline query problem*

2.2 Semantic Web

The original Web was designed to provide information in a human-readable format, which meant machines could not process or reason about the data. Queries asked to the Web (e.g., who performed this piece of music?) could not be answered. Although web mining tools were researched and proved useful in some situations, no extraction algorithm is robust enough to handle every domain. An approach to circumvent this limitation is to make the data explicit on the Web in a way which allows machines to process the information. This approach was the first step towards the *Semantic Web*. Tim Berners-Lee defines this term as follows:

The Semantic Web is not a separate web but an extension of the current web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation. [8]

Although the Extensible Markup Language (XML) [11] provides additional information about the data, it provides no semantics and thus is inflexible for machine processing. To overcome the limitations of XML, machine-processable semantics can be added to the data, which is the purpose of RDF and web ontologies [40].

2.2.1 RDF Data Model

Resource Description Framework (RDF) [29], a W3C recommendation, is a directed, labeled graph data model for asserting *statements* about *resources*. The Web is built around the concept of Uniform Resource Identifiers (URI) [7]. A URI can identify anything, from a document to a person, a performance, an audio signal, etc. Statements, or facts, are expressed in this data model using *triples*, where each triple consists of a *subject*, *predicate*, and an *object*. It is common to refer to an *RDF statement* as a *subject* having a *property name* whose value is the *property value*, e.g., object, which may be a literal or another resource. A set of triples may be depicted as a graph of resources whose edges represent the binary relationships between them. As an example, Figure 2.2 shows an RDF graph containing four RDF statements. In this example, the graph asserts the statement that “Richard Mutt is taking the course ‘Modeling Data with OWL’.” Using common notation, ellipses denote resources and squares denote literals. If a shorthand notation is used to reduce the full URI reference, `http://www.ncsu.edu.edu/data#` and `http://www.ncsu.edu.edu/addressbook#` can be represented as “d:” and “ab:”, respectively. This statement can be represented in triple notation (subject, predicate, object), as shown in Table 2.1.

2.2.2 SPARQL

RDF can be serialized into many different formats, such as XML [3], N3 [30], Turtle [4], and N-Triples [19]. Although the N3, Turtle, and N-Triples formats are often more concise and presented in a more human-readable format, the XML form of RDF, i.e., RDF/XML, is the preferred syntax. Several techniques for querying XML documents exist, but with the Semantic Web in mind, the interest is not in querying at the *syntactic* level but querying at the *semantic* level [14]. In order to query the Semantic Web, a query language that recognizes the semantics of RDF is needed. Several such languages

Table 2.1: Triple notation for Figure 2.2

(d:student1, ab:firstName, “Richard”)
(d:student1, ab:lastName, “Mutt”)
(d:student1, ab:takingCourse, d:course59)
(d:course59, ab:courseTitle, “Modeling Data with OWL”)

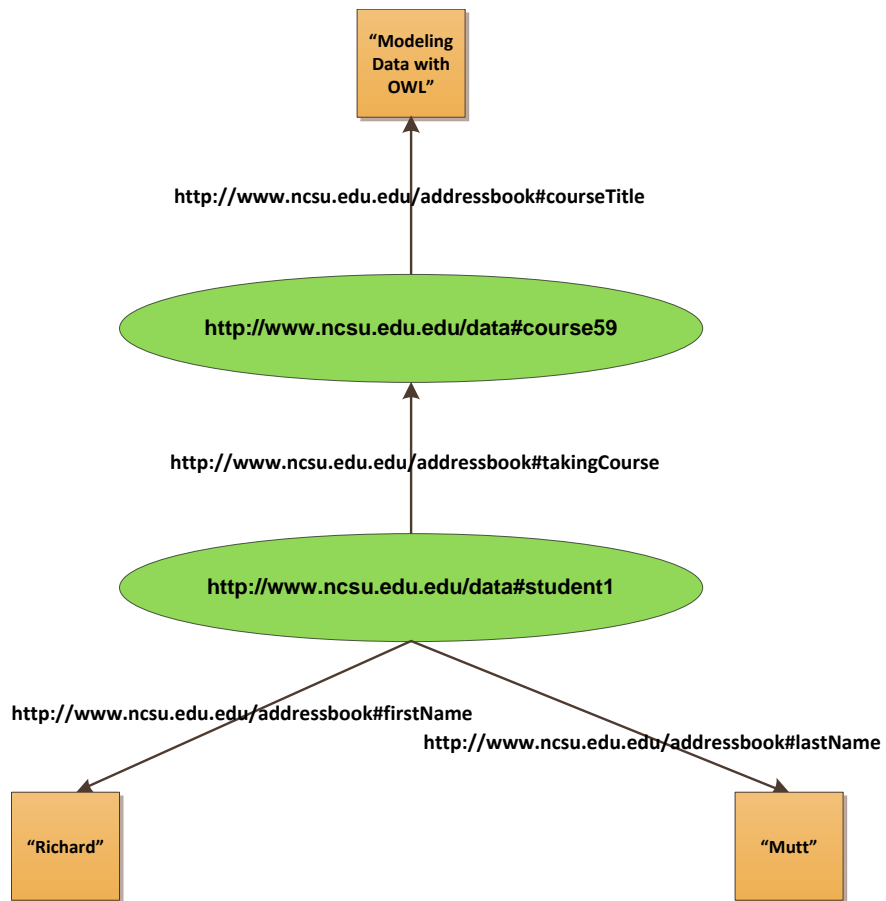


Figure 2.2: RDF Graph

exist [13][22][35][37][38], but we focus on SPARQL [21] (a recursive acronym for SPARQL Protocol and RDF Query Language) because it is the current W3C recommendation for querying RDF data.

Using the RDF graph in Figure 2.2 as an example, we wish to find which course(s) Richard Mutt is taking. One can answer this query by executing a simple SPARQL query, which has an SQL-like syntax, in the following manner, where variables are prepended with “?”.

```
SELECT ?course
WHERE { ?student ab:firstName "Richard".
        ?student ab:lastName "Mutt".
        ?student ab:takingCourse ?course.
}
```

2.3 Motivating Example

Now that some background material has been introduced, we can provide some more insight into our motivating example. Revisiting the shopping list example from Chapter 1, the shopper poses the following query in Query 2.3.1.

Query 2.3.1 *I want to purchase milk, eggs, and bread (one of each) from one or more stores such that the total price is minimized and the average rating of the stores is maximized.*

Figure 2.3 provides the data which this query uses. 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. When multiple preference criteria are specified as in our example (total price, average rating), there may be multiple possible optimal results rather than a single one so that the set of Pareto optimal solutions or “skyline” is computed. However, skyline queries have so far been considered for item queries and extensions are needed to support querying for packages. Given the example data in Figure 2.3, one possible package for our example scenario is *aee*, i.e., buying milk from store *a*, and eggs and bread both 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

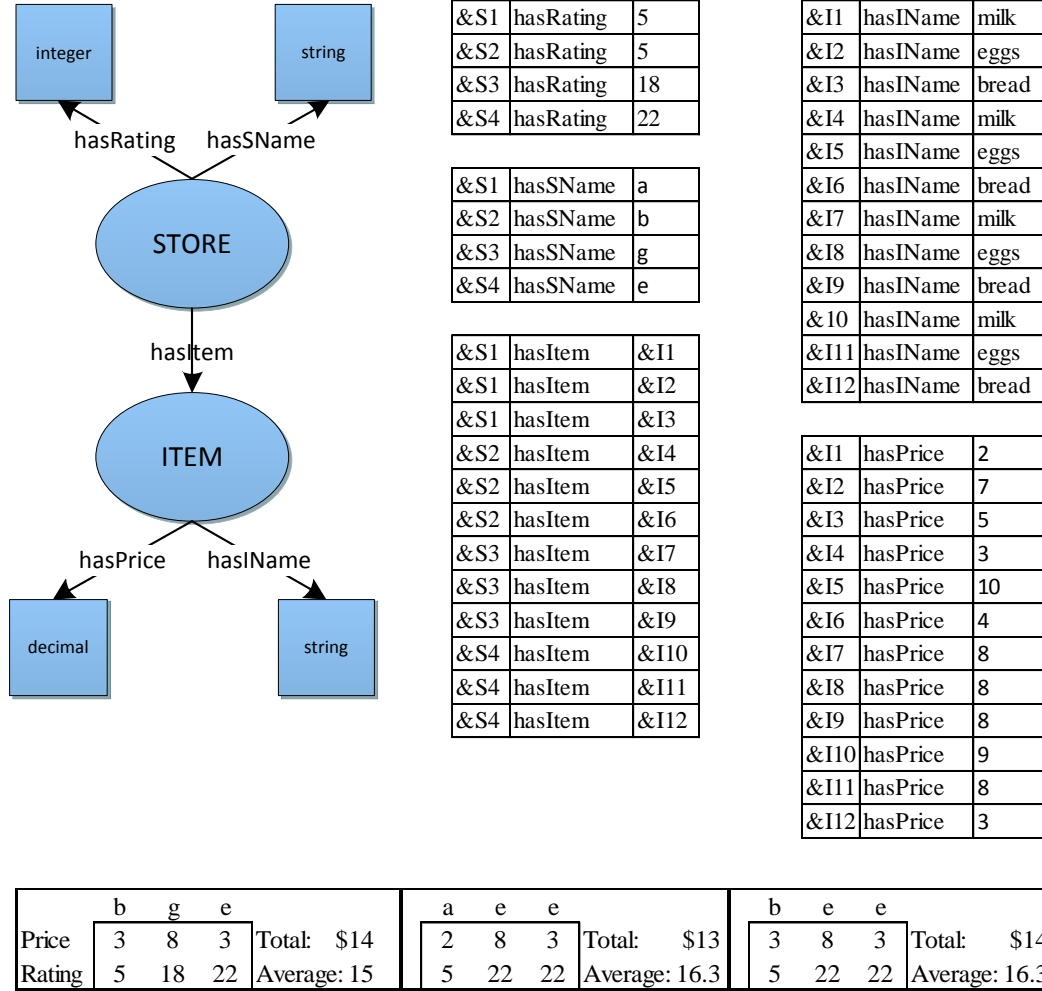


Figure 2.3: Data For E-Commerce Example

Figure 2.3 shows the total price and average rating for packages *ae*, *bee* and *bge*. We see that *ae* is a better package than *bee* because it has a smaller total price and the same average rating. On the other hand, *bge* and *ae* are incomparable because although *bge*'s total price is worse than *ae*'s, its average rating is better.

Specifying our purchasing 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 the list of desired products, *milk*, *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

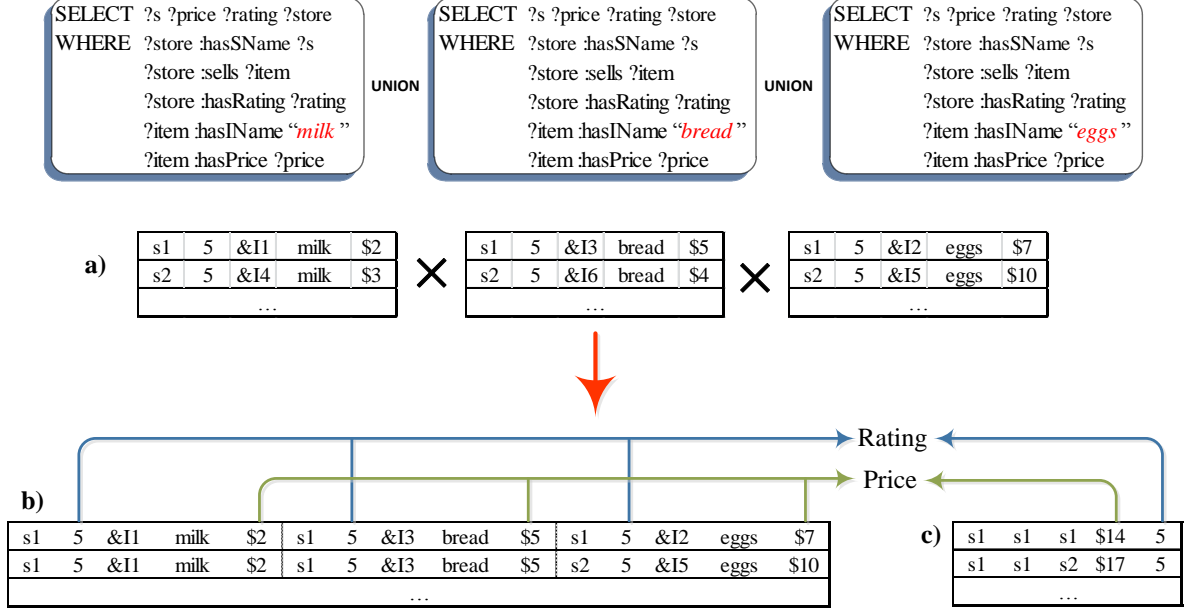


Figure 2.4: Dataflow for the SkyPackage Problem in Terms of Traditional Query Operators

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, as shown in Figure 2.4.

We now provide a more formal presentation of SkyPackage graph pattern queries.

2.4 Problem Definition

Let D be a dataset with property relations P_1, P_2, \dots, P_m and GP be a graph pattern with triple patterns TP_i, TP_j, \dots, TP_k (TP_x means triple pattern with property P_x). $[[GP]]_D$ denotes the answer relation for GP over D , i.e., $[[GP]]_D = P_i \bowtie P_j \bowtie \dots \bowtie P_k$. 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 ($?item, hasName, \text{"milk"}$) *target qualifying constraints* since those items determine the stores that are selected.

We review the formalization for preferences given in [32]. Let $Dom(a_1), \dots, Dom(a_d)$ be the domains for the columns a_1, \dots, a_d in a d -dimensional tuple $t \in [[GP]]_D$. Given a set of attributes $B \subseteq A'$, a preference PF over the set of tuples $[[GP]]_D$ 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, \dots, 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 2.4.1 (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 $GP = TP_1, \dots, TP_m$ and a set of preferences $PF = PF_i, PF_j, \dots, PF_k$, 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 2.4.2 (Map and Generalized Projection) *Let $\mathcal{F} = \{f_1, f_2, \dots, f_k\}$ 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 $\mathcal{X} = \{x_1, x_2, \dots, x_k\}$ with x_i generated by the function f_i in \mathcal{F} .*

Generalized Projection $\prod_{colr_x, colr_y, colr_z, \dots, \hat{\mu}_{[\mathcal{F}, \mathcal{X}]}}(R)$ *returns the relation $R'(colr_x, colr_y, colr_z, \dots, x_1, x_2, \dots, x_k)$. 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 2.4.3 (SkyPackage Graph Pattern Query) . *A SkyPackage graph pattern query is graph pattern $GP_{[\{c_1, c_2, \dots, c_N\}, \mathcal{F}=\{f_1, f_2, \dots, f_k\}\{PF_{P_i}, PF_{P_j}, \dots, PF_{P_k}\}, r]}$ 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 2.4.4 (SkyPackage Query Answer) *The answer to a SkyPackage graph pattern query R_{SKY} can be seen in the result of the following steps:*

1. $R_{product} = [[GP_{c_1}]] \times [[GP_{c_2}]] \times \dots \times [[GP_{c_N}]]$ such that $[[GP_{c_x}]]$ is the result of evaluating the branch of the union query with constraint c_x . Figure 2.4.(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. $R_{project} = \prod_{r_1, r_2, \dots, r_N, \hat{\mu}[\mathcal{F}=\{f_1, f_2, \dots, f_k\}, \mathcal{X}=\{x_1, x_2, \dots, x_k\}]} (R_{product})$ where r_i is the column for the return variable in subquery i 's result, $f_1 : (\text{dom}c_1(o_1) \times \text{dom}c_2(o_1) \times \dots \times \text{dom}c_N(o_1)) \rightarrow \mathbb{R}$ where $\text{dom}c_1(o_1)$ 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 Figure 2.4.(c).
3. $SKYLINE[R_{project}, \{PF_{P'_1}, PF_{P'_2}, \dots, PF_{P'_k}\}]$ 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.

2.5 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.

2.5.1 Historical Perspective

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 are unfit for databases because the solutions assume all data resides in main-memory, which is limited, and is inefficient for large data sets.

2.5.2 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][39], which has to read the whole database at least once, index-based algorithms [26][33] allow one to access only a part.

2.5.3 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.

[41] 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 meets 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. [32] 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. [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.

2.5.4 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 [42][43]. [43] 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 [42]. 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.

Chapter 3

Techniques for Evaluating Skyline Packages

We present in this chapter four approaches to solving the package skyline problem.

3.1 Algorithms for Package Skyline Queries over Vertically 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].

3.1.1 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 Figure 2.3. Solving the skyline package problem using *JCPS* involves the following steps:

1. $I \leftarrow \text{hasIName} \bowtie \text{hasSName} \bowtie \text{hasItem} \bowtie \text{hasPrice} \bowtie \text{hasRating}$

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
		⋮	

Figure 3.1: 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
				⋮			

Figure 3.2: Cartesian Product Result

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 Figure 3.1, 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 Figure 3.2. 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 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.

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

Algorithm 1: JCPS

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

Output: Package Skyline \mathcal{P}

```
1:  $n \leftarrow$  package size
2:  $I \leftarrow VPT_1 \bowtie VPT_2 \bowtie \dots \bowtie VPT_x$ 
3: for all  $i \in [1, n - 1]$  do
4:    $I \leftarrow I \times I$ 
5: end for
6: for all  $i \in [1, y]$  do
7:    $I \leftarrow \mathcal{A}_{s_i}(I)$ 
8: end for
9:  $\mathcal{P} \leftarrow$  skyline( $I$ )
10: return  $\mathcal{P}$ 
```

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.

3.1.2 *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 3.1.1 *Let $\rho = \{p_1 p_2 \dots p_n\}$ and $\rho' = \{p'_1 p'_2 \dots p'_n\}$ be packages of size n and ρ is 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, \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 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 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, $p_1[x_1] + p_2[x_1] \preceq p_1[x_1] + p'_2[x_1]$ ($3 + 3 = 6 \preceq 7 = 3 + 4$). Since all attribute values except $p'_2[x_1]$ remained unchanged, by definition of skyline we conclude $\rho \preceq \rho'$.

This lemma suggests that we can modify *JCPS* by introducing a skyline phase before the Cartesian product step as a way of pruning input. 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 sorted VPTs each containing a skyline attribute. We call this algorithm *RSJFH – CPS*. The 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. The overhead of these additional phases could diminish any benefits of pruning using skyline results and will unlikely be much better, if at all, than *JCPS*.

To illustrate *RSJFH – CPS*, given the VPTs *hasIName*, *hasSName*, *hasItem*, *hasPrice*, and *hasRating* obtained from Figure 2.3, solving the skyline package problem involves the following steps:

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

hasName ✕ hasPrice				hasSName ✕ hasItem ✕ hasRating					
&l1	milk	&l1	2	&S1	A	&S1	&l1	&S1	5
&l4	milk	&l4	3	&S1	A	&S1	&l2	&S1	5

Figure 3.3: *RSJFH* (skyline-over-join) for milk

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

Figure 3.4: *RSJFH*'s result for milk

Figure 3.3 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 Figure 3.4. 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 Figure 3.5, which produces a table similar to the one in Figure 3.2. 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 lines 8-10. 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 7), each target table I_i^1 along with I^2 is sent as input to *RSJFH* for a skyline-over-join operation. Afterwards, all resulting target tables

item	price	store	rating		item	price	store	rating		item	price	store	rating
milk	2	A	5	✕	eggs	3	B	5	✕	bread	5	A	4
⋮					⋮					⋮			

Figure 3.5: Cartesian product on all targets (e.g., milk, eggs, and bread)

Algorithm 2: RSJFH-CPS

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

Output: \mathcal{P}

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

undergo a Cartesian product phase (line 12) to produce all possible combinations, and then all equivalent attributes are aggregated (lines 13-15). Lastly, a single-table skyline algorithm is performed to discard non-skyline packages.

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*. By observing these steps and using Lemma 3.1.1 as a guide, a more efficient algorithm can be devised that exploits this lemma more productively.

3.2 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.

MILK		EGGS		BREAD		RATING	
store	price	store	price	store	price	store	value
a	2	g	5	e	3	e	5
b	3	a	7	b	4	i	5
f	3	c	8	a	5	c	12
e	4	e	8	i	8	f	13
g	6	h	9	g	5	h	14
e	9	b	10	f	6	g	18
h	9	d	10	h	6	a	20
d	10			d	10	d	21
						b	22

Figure 3.6: E-commerce Data

3.2.1 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 can be seen to be 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 (*hasPrice*) 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 Figure 3.6 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 needed 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 Figure 3.6 shows the target descriptive table for our example. The tables can easily be constructed using appropriate queries using a storage layout that provides an efficient support for joins.

3.2.2 Notations

In general, the build phase produces a set of partitioned tables $T_1, \dots, T_n, T_{n+1}, \dots, T_m$, 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, \dots, T_n are the target qualifying tables where n is the number of qualifying constraint. T_{n+1}, \dots, 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.

Definition 3.2.1 (Local and Global constraints) *If a constraint involves either a target attribute or a target qualifying attribute (e.g., milk's price > \$5), we call this constraint a local constraint. Local constraints can be evaluated by looking at the relevant tables. If a constraint involves an aggregation over the package's attribute (e.g., total price < \$10), we call this constraint a global constraint. Global constraints can be evaluated only after the packages are formed.*

Depending on the data at hand, there is a property of the storage model that may hold, but does not affect the approach.

Property 3.2.2 $\bigcap_{i=1}^n T_i^1 = \emptyset$

This property states that it is possible that each target qualifying table consists of unique elements. Revisiting the shopping list problem, for example, it may be possible that a store sells only one item. Though this property is optional, the following property will always hold.

Property 3.2.3 $\bigcup_{i=1}^n T_i^1 \subseteq T_l^1$, where $l \in [n + 1, m]$

This property ensures us that all elements in the target qualifying tables are in the target descriptive tables. As an example, in Figure 3.6, the *rating* table contains all stores that sell all three items. In this example, the property yields an equality. In other examples, this property may yield a proper subset. That is, the *rating* table may contain other stores that do not sell the desired items but other items outside of the query.

CPJS and SkyJCPS Algorithms

Given the 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*). Not only is the time complexity inefficient, but also the space complexity. Given n targets and m target qualifiers, n^m possible combinations exist as an intermediate result prior to performing a skyline algorithm. Since performing a Cartesian product has a time complexity of $\mathcal{O}(n^m)$, the time complexity of determining the package skyline is at least this, regardless of which skyline algorithm is chosen. 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 would have to be performed such as aggregations, which have 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: $T_1, T_2, \dots, T_n, T_{n+1}, \dots, T_m$

Output: \mathcal{P}

```
1:  $I \leftarrow T_1 \times T_2 \times \dots \times T_n$ 
2: for all  $i \in [n + 1, m]$  do
3:    $I \leftarrow I \bowtie T_i$ 
4: end for
5:  $\mathcal{S} \leftarrow \text{skyline}(I)$ 
6: return  $\mathcal{S}$ 
```

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 one table (line 3). Finally, a single-table skyline algorithm is performed to eliminate dominated packages (line 5).

Applying this approach to the data in Figure 3.6, one would have to compute all 448 possible combinations before performing a skyline algorithm. We show next how the size of this intermediate result can be drastically reduced.

Using the intuition given in § 3.1.2, an initial skyline phase can be introduced on each target, e.g., milk, before the Cartesian product phase. Although similarities to *RSJFH – CPS* can be observed, *SkyJCPS* yields better performance because of the reduced number of joins. Figure 3.3 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 Figure 3.6, solving the skyline package problem involves the following steps:

1. For each target qualifying table TQ_i (e.g., milk)

(a) $I_{TQ_i} \leftarrow (TQ_i) \bowtie \textit{rating}$

(b) $I'_{TQ_i} \leftarrow \textit{skyline}(I_{TQ_i})$

2. $CPJS(I'_{TQ_1}, \dots, I'_{TQ_i}, \textit{rating})$

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.

3.2.3 SkyPackage Algorithm

We approach the skyline package problem similar to a divide-and-conquer problem except that the problem is given already divided. That is, each target is represented as separate tables. We conquer, i.e., find the skyline, each of these tables and then cross-product, i.e., combine to produce intermediate data before performing a skyline algorithm.

Our algorithm attempts to discern the skyline set by stepping through sorted sublists 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 Figure 3.6, 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 Figure 3.7, $\{(age), \$10, 14\}$ and $\{(bbb), \$17, 22\}$. All future package skylines must fall between these two packages. We

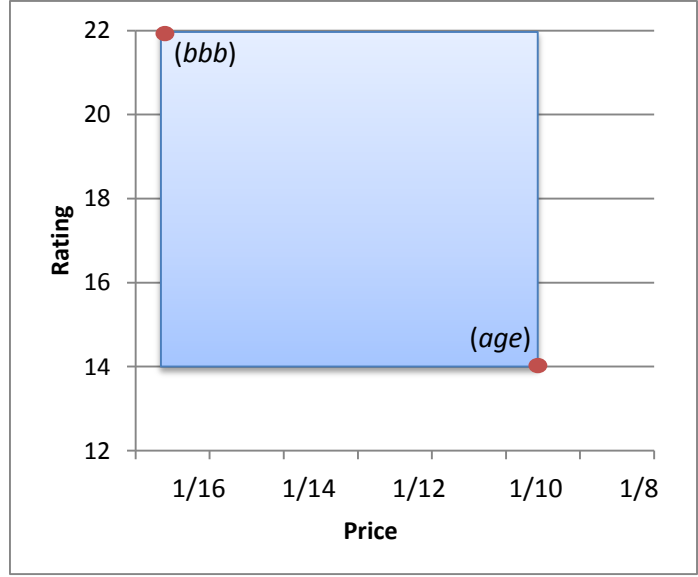


Figure 3.7: Skyline Region

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 Figure 3.6. We present in Figure 3.2.4 four iterations where each iteration indicates a new tuple being examined. As we look at each store in the *milk* table, we probe 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 $(f, 3)$

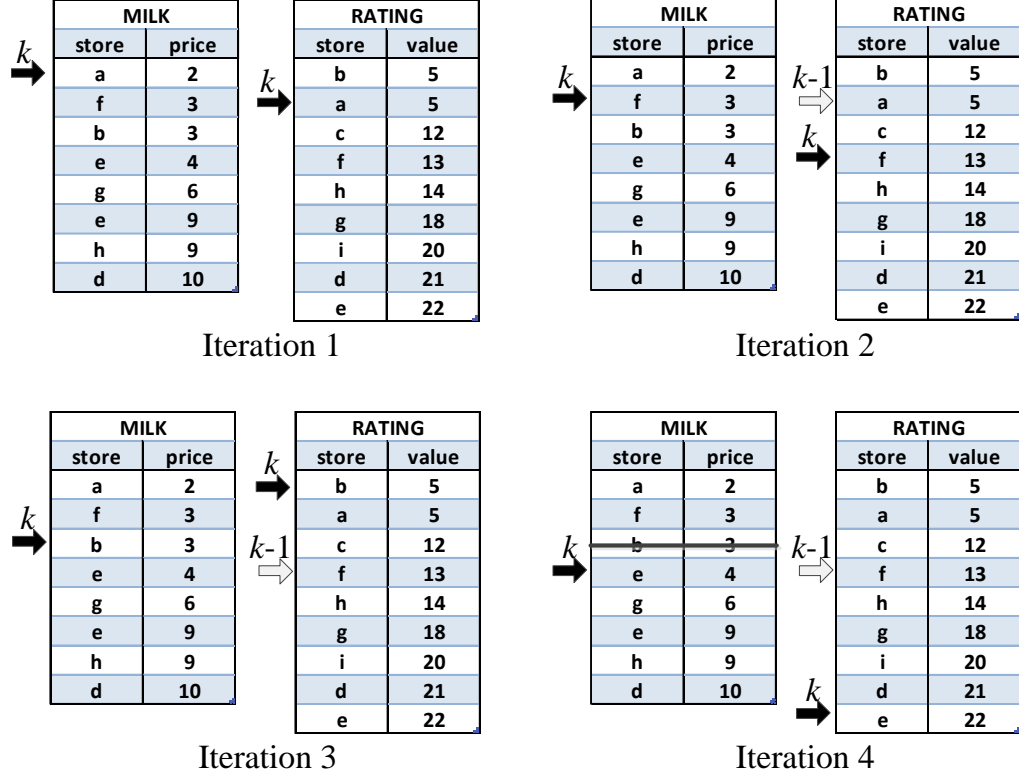


Figure 3.8: Pruning Example

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 3.2.4 (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. \square

Lemma 3.2.4 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 reduced to $|T_1| \times |T_2| \times \dots \times (|T_i| - 1) \times \dots \times |T_n|$. To illustrate this, after tuple $(b, 3)$ is pruned in Figure 4, the Cartesian product size is reduced from 448 tuples 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 3.2.4, we denote the current object as k and the previous object as $k-1$. If Lemma 3.2.4 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 3.2.5 (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 , $T_i[k] \preceq T_i[k+1] \preceq T_i[k+2] \dots$. Therefore, every object after k is dominated. \square

Lemma 3.2.5 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 Figure 3.2.4, 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 3.9 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.

MILK		EGGS		BREAD	
store	price	store	price	store	price
a	2	g	5	e	3
b	3	a	7	b	4
		b	10		

Figure 3.9: 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

Figure 3.10: Skyline Package Result

In Figure 3.9, 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 Figure 3.10.

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 a 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 3.2.4 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 3.2.5. 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.

Algorithm 4: SkyPackage

```

1:  $v_k(i) \leftarrow$  the value of object  $k$  in table  $i$ 
2:  $k \leftarrow$  the current object (i.e., row)
3:  $ptr \leftarrow$  the best object
4: for all  $i \in [1, n]$  do
5:    $ptr \leftarrow v_x(n + 1)$ ,  $x \leftarrow$  first tuple in  $i$ 
6:   for all  $k \in t_i$  do
7:     check whether Lemma 3.2.4 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 3.2.5 holds
19:     check  $k$  against local constraints
20:   end for
21: end for
22: cross product, remove tuples not satisfying global constraints
23: skyline

```

Chapter 4

Sesame Integration Framework

In this section, we present our implementation strategy for a loose integration with an open source RDF engine, Sesame 2, using a Web-based interface. We first provide an overview of Sesame’s architecture followed by the environment setup. Next, we discuss the client-server configuration used, and how we utilized Sesame’s server to query over HTTP.

4.1 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. For querying data, Sesame offers SPARQL and SeRQL (Sesame RDF Query Language) [13] querying and is extensible to other query languages.

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.

Sesame implements a multi-layered architecture whose components are shown in Figure 4.1. Since our framework is designed using HTTP, we exclude from our discussion the *Sail API*, *Rio*, and *SailRepository* components. We next highlight the main components that are pertinent in our framework.

- The *RDF Model* provides implementation and an interface for all RDF entities (e.g., statements, literals, etc.).

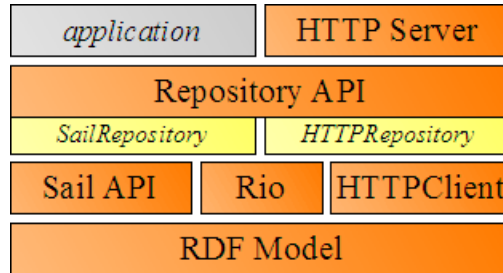


Figure 4.1: Sesame Components ¹

- The *Repository API* is a communication interface that offers a developer-friendly API for accessing (e.g., querying, manipulating, etc.) the data.
- The *HTTPRepository* component is an implementation of the *Repository API* and offers a client-server communication over HTTP with the Sesame server.
- The *HTTP Server* component provides access to the repositories over HTTP using Java Servlets and requires a compatible Web container (e.g., Apache Tomcat).

4.2 Framework

In this section, we present an overview of our framework and a description of how we stored and retrieved the data.

The data that was used in the framework was the MovieLens² dataset, which was converted to RDF format using the Jena API [15]. Its ontology is depicted in Figure 5.6. Figure 4.3 provides a high-level overview indicating the main components of our framework. We used a server with Linux and an Apache Tomcat Web container. 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 thesis, we chose to support a query of the following form.

Query 4.2.1 *Find packages of n movies such that the average rating of all the movies is high, average release date is high, and each movie-rater has rated at least one of the movies.*

¹Borrowed from <http://www.openrdf.org/doc/sesame2/users/ch03.html>

²<http://www.grouplens.org/node/73/>

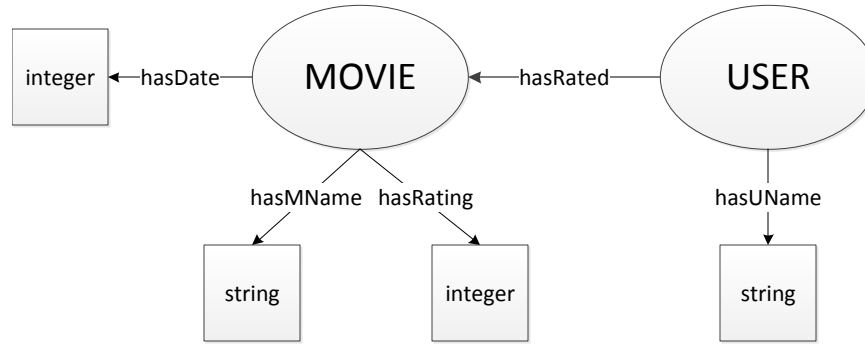


Figure 4.2: MovieLens Ontology

When the user provides preferences using the Web-based interface, the information is sent to the SPARQL adapter, which 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.

4.2.1 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*.

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*.

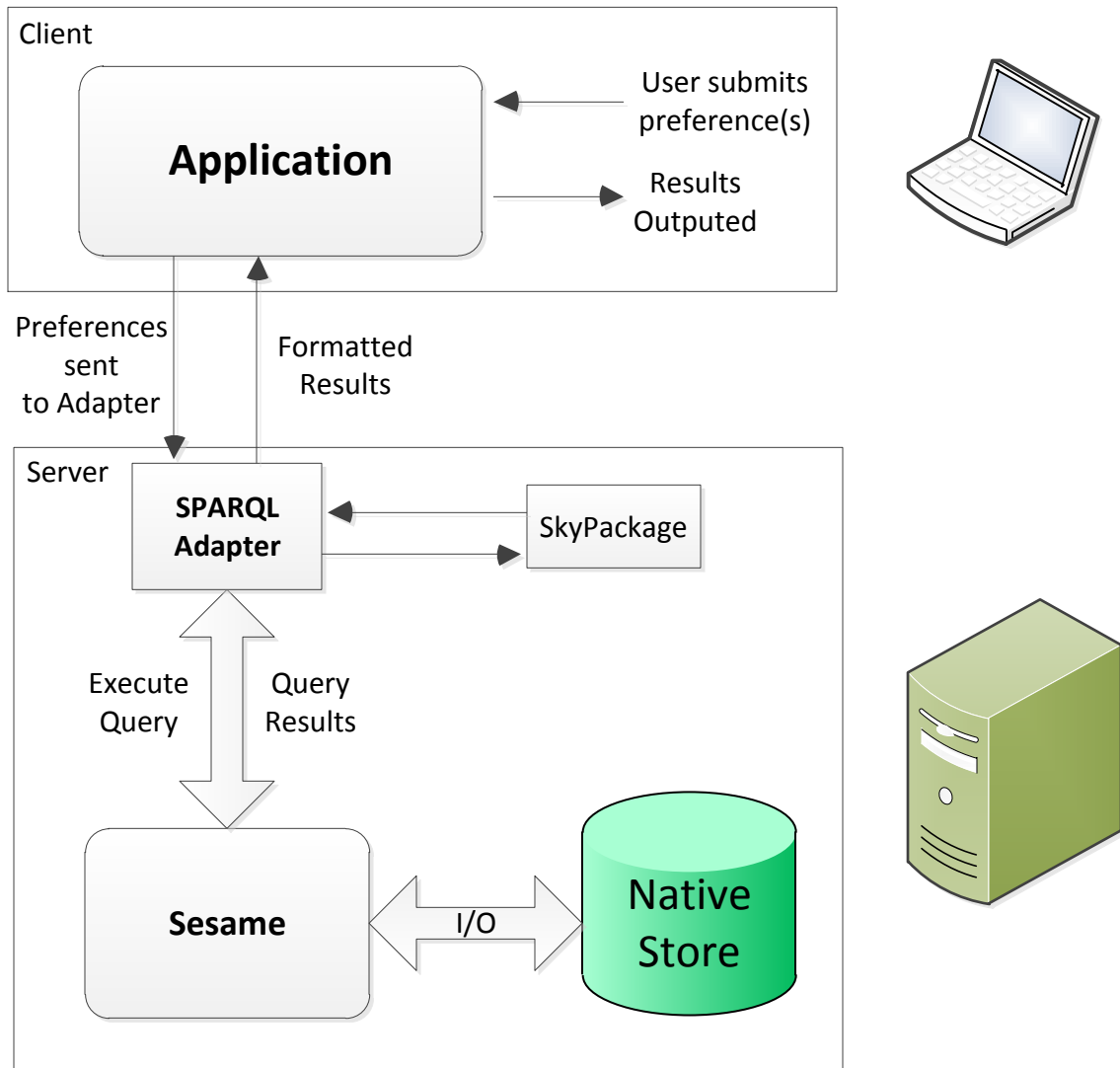


Figure 4.3: Framework with SPARQL Adapter

4.2.2 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 (§ 3.2.1). 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: $vpt_1, vpt_2, \dots, vpt_m, vpt_{m+1}, vpt_n$. Moreover, for the purpose of illustration, we assume vpt_1, \dots, vpt_m and vpt_{m+1}, \dots, vpt_n are sufficient to construct the TQ and TD tables, respectively. The arguments for the SP operator are defined as follows:

- $C = (c_1, c_2, \dots, c_m)$, 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 (as described in § 2.4) for each constraint. Assuming vpt_1 contains data to which a constraint can be applied, the target qualifying triple pattern is specified as $(?var :vpt_1 c_i)$, 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, c1)`.

To illustrate this process, consider Query 4.2.1 and the ontology depicted in Figure 5.6. Suppose the person issuing the query is interested in the following movie-raters: *user8* and *user34*. 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 Figure 5.6, we have the

```

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. }

```

Figure 4.4: Queries to construct target qualifying tables

```

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

```

Figure 4.5: Query to construct target descriptive table

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 Figure 4.4 are constructed, which produces the TQ tables

A similar 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 clause 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 Figure 4.5 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 *HTTPRepository* 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.

Chapter 5

Evaluation

In this chapter, we present the experiments conducted and the results of those experiments. We divided our experiments into two sections based on the data used, synthetic data and real data. We also provided an additional section that evaluates our storage model.

5.1 Overview

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 [15] and stored in Oracle Berkeley DB using B-tree indexes. All algorithms assumed data is initially stored on disk, and each tuple is brought into memory only when it is scanned.

We compared four algorithms, *SkyPackage*, *JCPS*, *SkyJCPS*, and *RSJFH* (*RDF-SkyJoinWithFullHeader*) [16]. During the skyline phase of each of these algorithms, we used the *block-nested-loops* (*BNL*) [10] algorithm. *RSJFH* is similar to *SkyJCPS* in that a skyline algorithm is performed before the Cartesian product. However, *RSJFH* is dependent upon the data being vertically partitioned, which increases the number of joins before the Cartesian product can be performed (see § 3.3). *RSJFH* incorporates the join with knowledge of the skyline, thereby decreasing the number of tuples entering into the Cartesian product phase.

We compared these algorithms using two types of scalability metrics: scalability of data size and scalability of package size.

5.2 Synthetic Data

5.2.1 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 [10]. 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.

5.2.2 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 5.1 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, Figure 5.1 indicates that this triple size yields approximately 52.5M 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. Figure 5.2 and Figure 5.3 show the results and are plotted using a logarithmic scale. No anomalies were found within packages of the same size. However, due to the logarithmic scale used, it seems that some of the algorithms have the same execution time as the triple size increases. This is not the case, and all algorithms' execution increased as the data size increased.

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
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

Figure 5.1: Synthetic Data Triple Sizes

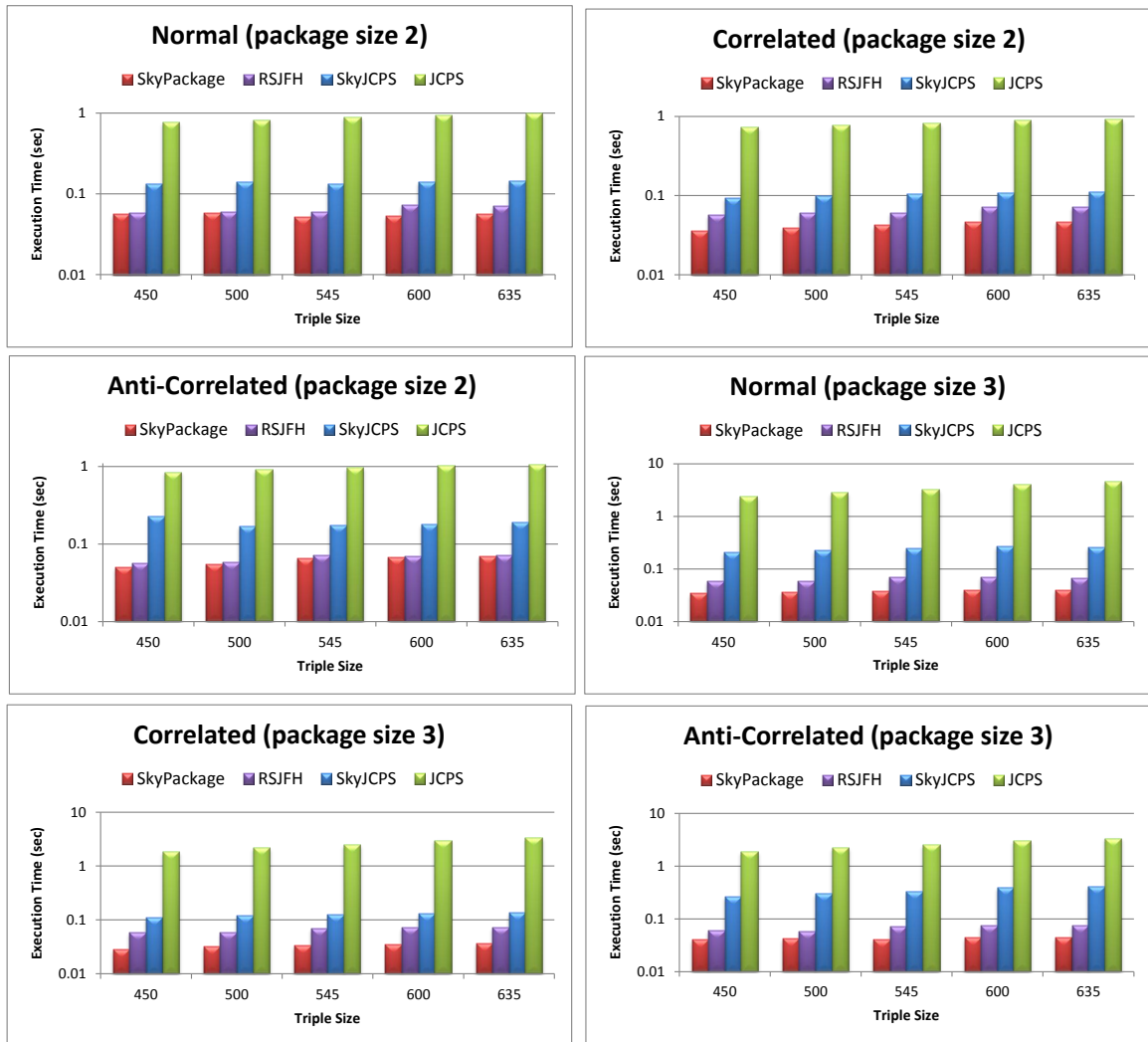


Figure 5.2: Evaluation Results for Package Sizes 2 and 3

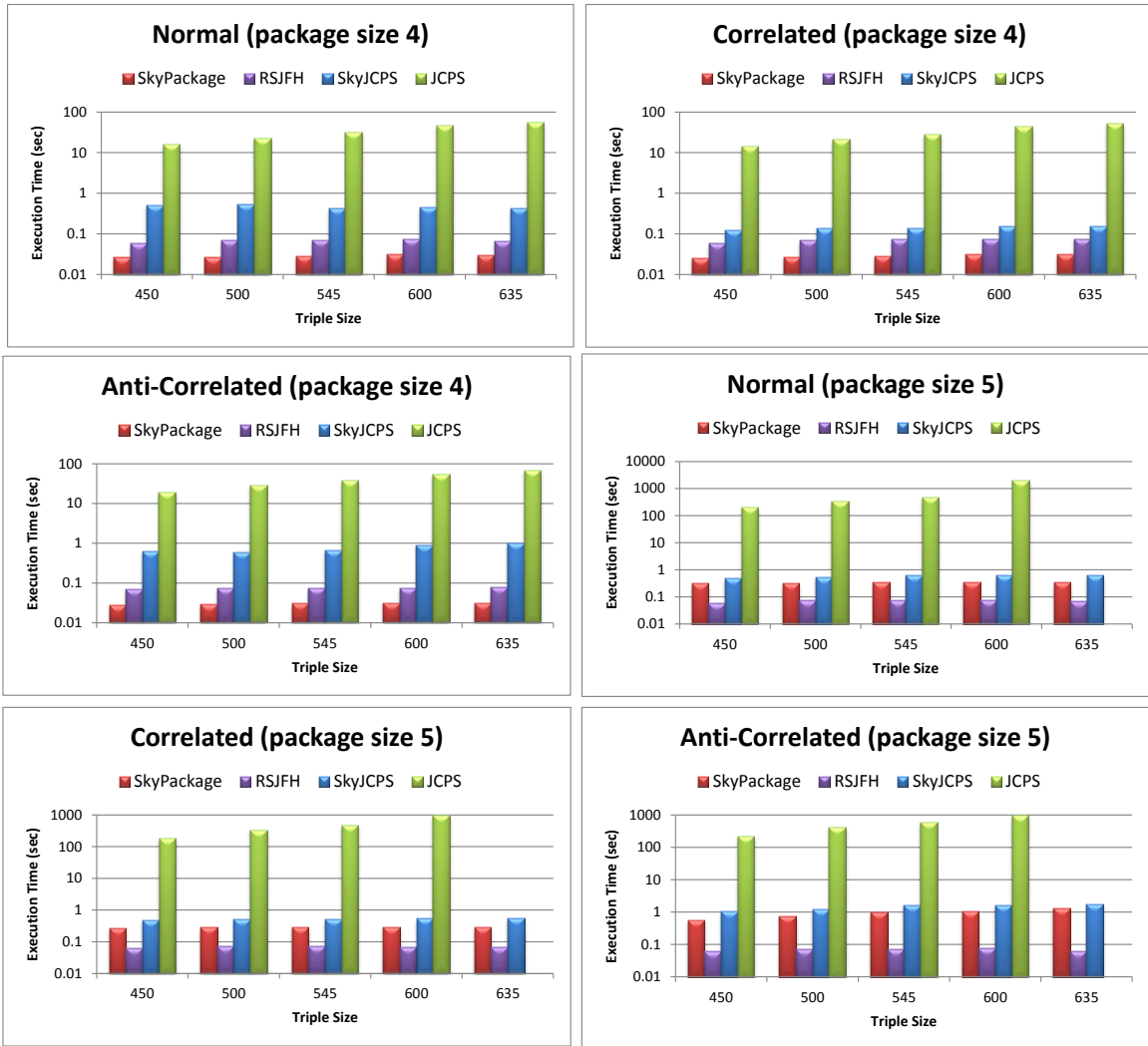


Figure 5.3: Evaluation Results for Package Sizes 4 and 5

5.2.3 Package Size Scalability

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. In Figure 5.4 we also show how the algorithms perform across packages of size 2 to 5 for the same triple size. For this experiment, we chose the largest triple size of 635, which yields the longest running time. 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. Unlike *SkyPackage*, which uses early termination, *SkyJCPS* has to examine every tuple and this adds to the total execution time. In addition, *SkyPackage* utilizes a pruning technique that further reduces the Cartesian product’s input size. Note that there is a larger increase in execution time between package size 4 and package size 5 than there is between the other packages. This increase is merely due to the larger increase in size of the Cartesian product, as can be seen in Figure 5.1. Even though the tuples go through a pruning phase prior to entering the Cartesian product phase, the pruning power deteriorates as the package size increases, which is discussed in the next section.

RSJFH outperformed *SkyPackage* for packages of size 5. We argue that *SkyPackage* may perform slightly slower than *RSJFH* on small datasets distributed among many tables. In this scenario, *SkyPackage* has six tables to examine, while *RSJFH* has only two tables. Evaluation results from the real datasets (later in this chapter) ensure us that *SkyPackage* 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 as the triple size increases. This is not the case, and all algorithms’ execution increased as the data size increased.

5.2.4 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 Figure 5.5. 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

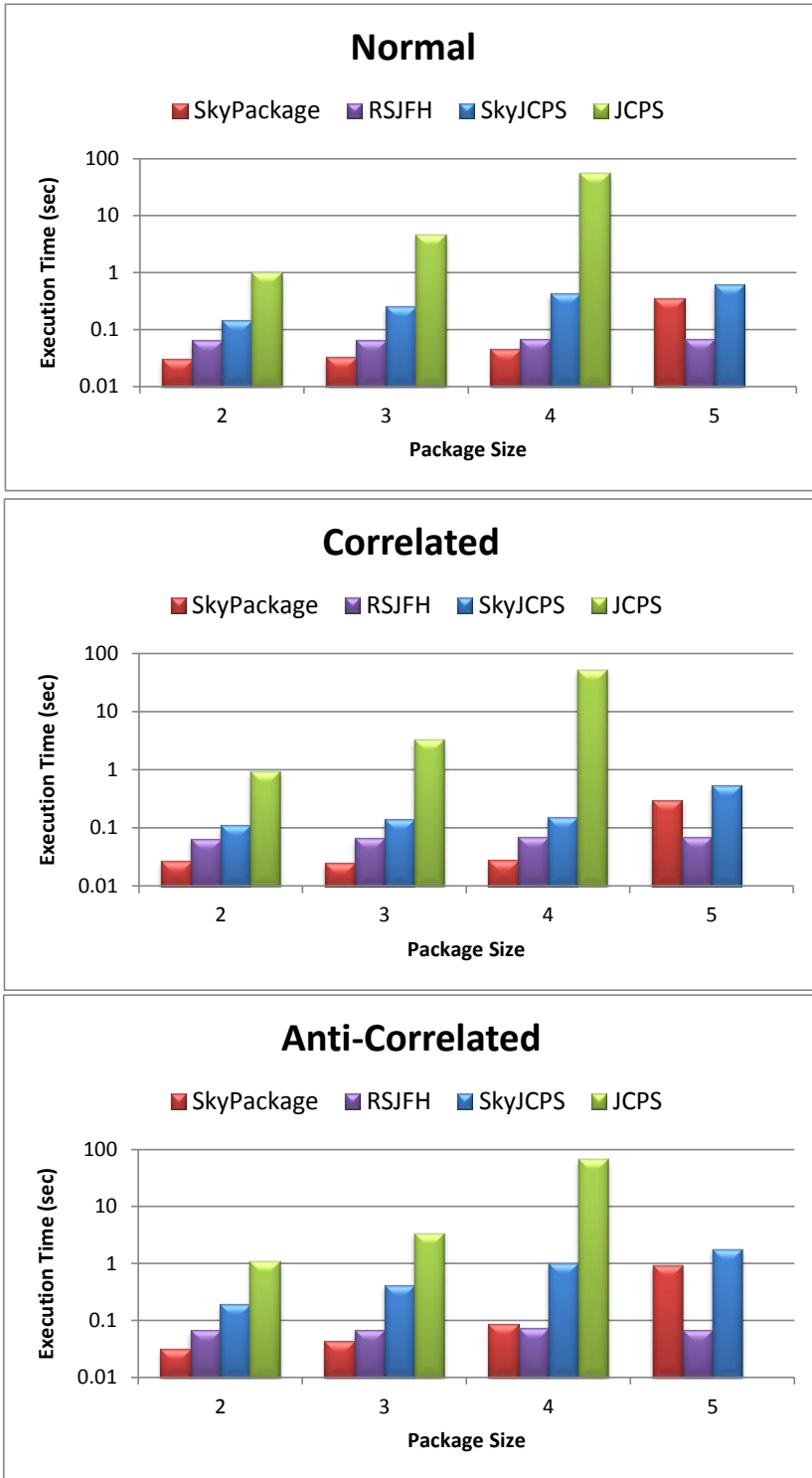


Figure 5.4: Scalability for Package Sizes 2 to 5

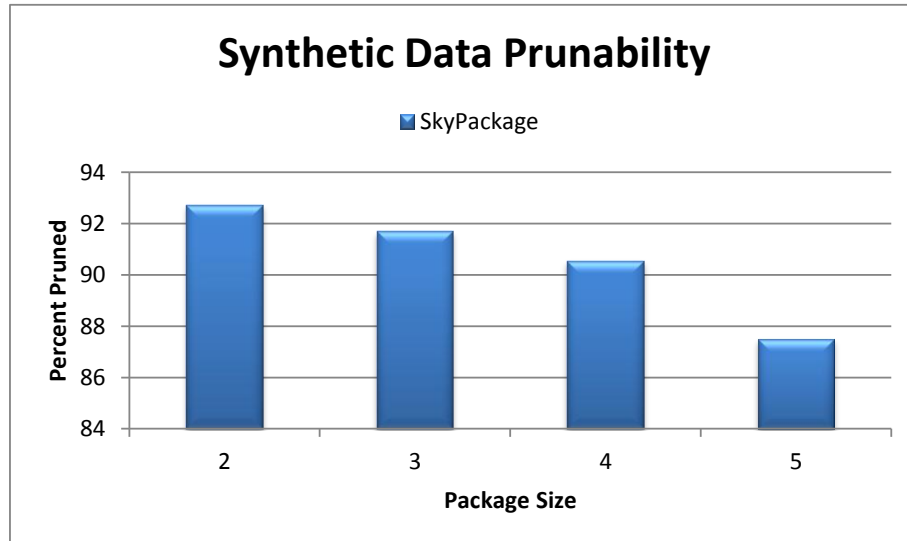


Figure 5.5: Prunability of Synthetic Data

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). 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*.

5.3 MovieLens Dataset

The first real-world dataset evaluated was MovieLens¹, which consisted of 10 million ratings and 10,000 movies.

¹<http://www.grouplens.org/node/73/>

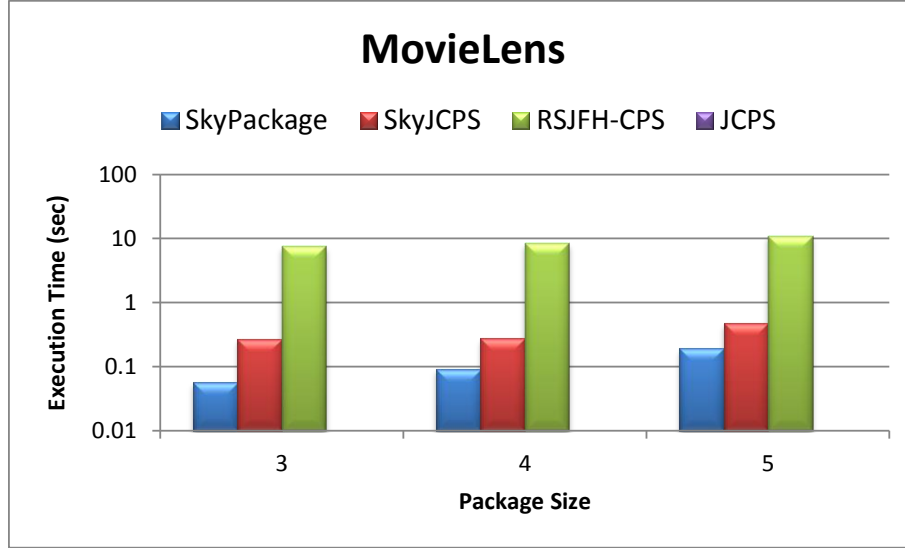


Figure 5.6: Package Size Scalability for MovieLens

5.3.1 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 4.2.1, 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.

The results of this experiment can be seen in Figure 5.6. 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. Due to the logarithmic scale used, it may seem that some of the algorithms have the same execution time as the triple size increases. This is not the case, and all algorithms' execution increased as the data size increased.

Figure 5.7 shows the percent of tuples pruned by *SkyPackage*. In all three package sizes, approximately 99% of the tuples were pruned.

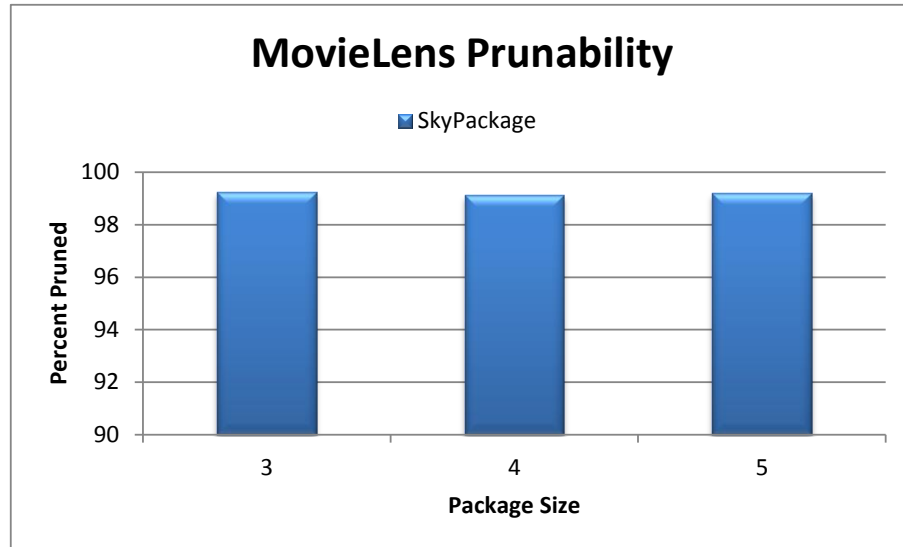


Figure 5.7: Prunability of MovieLens Dataset

5.4 Book-Crossing Dataset

The next real dataset used for evaluations was Book-Crossing², which consists of approximately 271,000 books rated by approximately 278,000 users.

5.4.1 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, 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 5.4.1, where $n = 3, 4, 5$ for evaluation.

Query 5.4.1 *Given n book-raters, find one or more 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.*

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

²<http://www.informatik.uni-freiburg.de/~chiegler/BX/dataset>

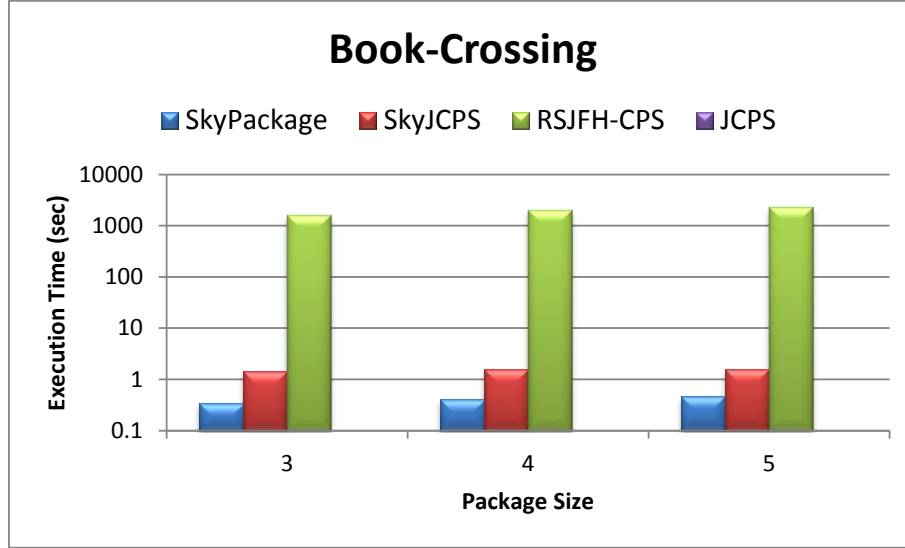


Figure 5.8: Package Size Scalability for Book-Crossing

included all five users.

The results of this experiment can be seen in Figure 5.8. 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. Due to the logarithmic scale used, it may seem that some of the algorithms have the same execution time as the triple size increases. This is not the case, and all algorithms' execution increased as the data size increased.

Figure 5.9 shows shows the percent of tuples pruned by *SkyPackage* from the Book-Crossing dataset.

5.5 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 vertical partitioned tables (VPT).

For the synthetic data, we took the average time over the three data distributions of

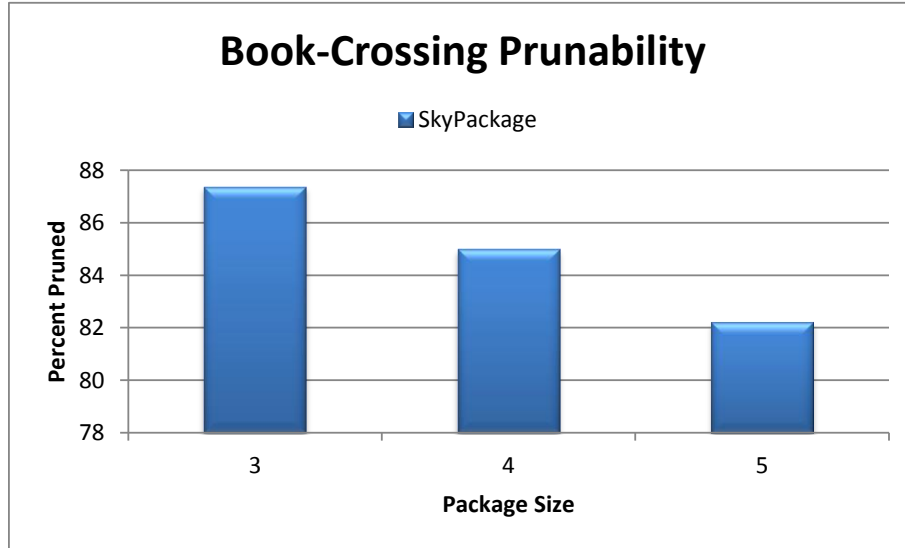


Figure 5.9: Prunability of Book-Crossing Dataset

the same package size. Figure 5.10 shows the results for packages of size 2, 3, 4, and 5. In Figure 5.11 and Figure 5.12, 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 § 3.2. 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.

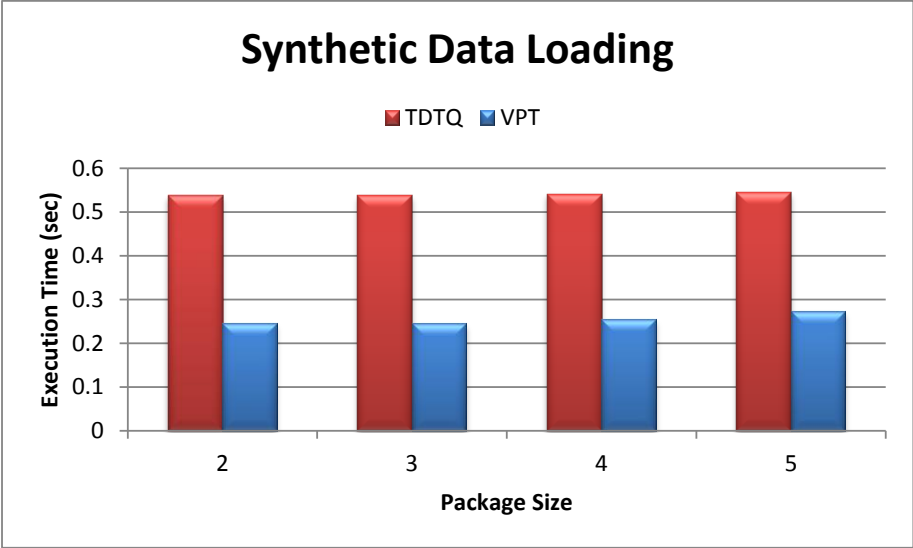


Figure 5.10: Database build for Synthetic Data

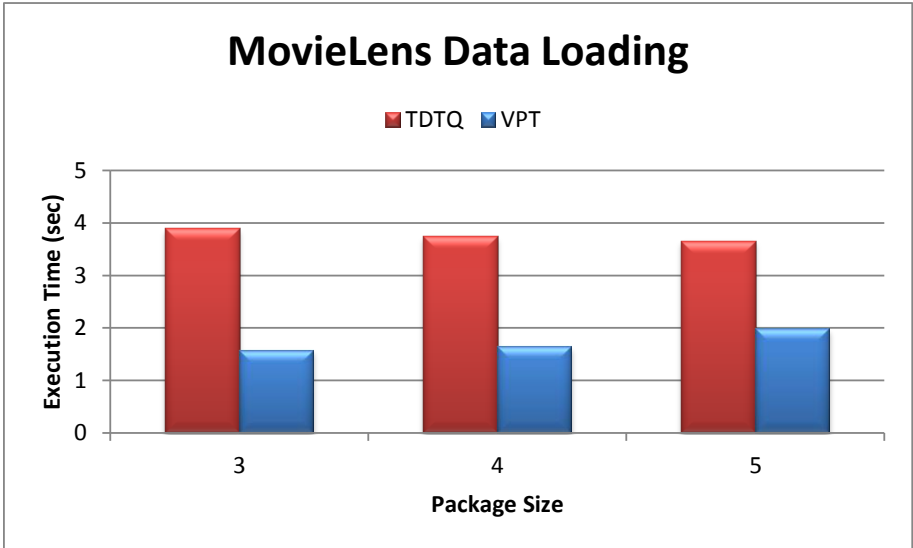


Figure 5.11: Database build for MovieLens Data

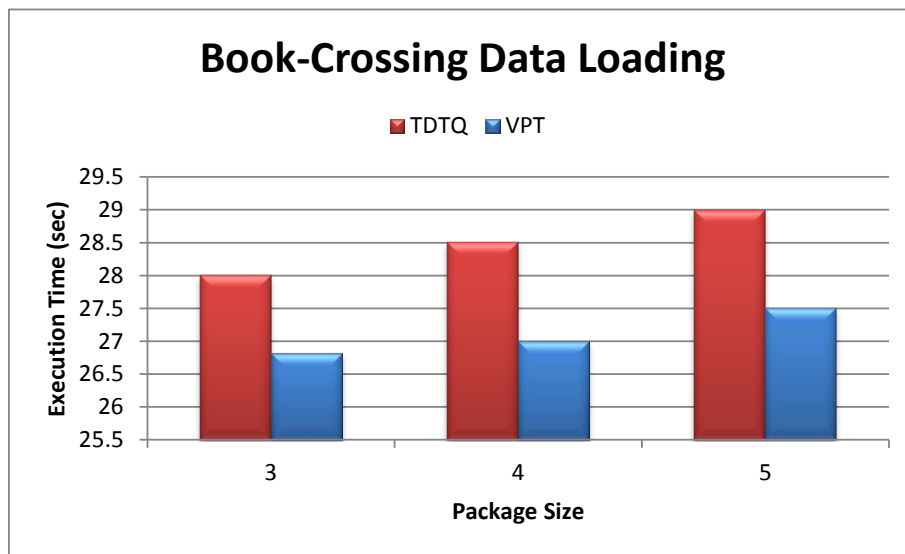


Figure 5.12: Database build for Book-Crossing Data

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis addressed the problem of answering skyline package queries. We have formalized and described what constitutes a “package” and common applications where this concept arises. We proposed 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, and have defined the term *skyline packages*. Package querying is especially useful for cases where a user requires multiple objects to satisfy certain constraints, as users are often interested in a collection of items or “packages” such that each package collectively satisfies a single need. Since finding such packages using the traditional item querying model is not feasible, a new approach needs to be taken.

We introduced SkyPackage, a skyline-based algorithm for answering queries involving packages. SkyPackage utilizes an extremely powerful pruning capability that lessens the number of combinations that need to be formed when aggregating over attributes. We also presented a loose integration of *SkyPackage* and Sesame, an open source RDF database, to show its feasibility and how it can be used in the real world. Finally, we presented various experimental evaluations that show its effectiveness using synthetic datasets and two real datasets, MovieLens and Book-Crossing.

6.2 Future Work

From our experience while conducting research, we discovered how our approach might be improved and used in other areas. The following suggests possible directions for future work.

- Our work has its limitations in answering queries involving a preference that depends on the combination, such as distance (e.g., total distance ≤ 50 miles). While we devised a simple solution to overcome this, it was not feasible in real-world applications.
- The research presented in this thesis may provide groundwork for providing such techniques to construct a more intelligent e-learning system.
- *SkyPackage* performance might be improved by providing a heuristic based on the combinations of previous skyline packages.
- Top- k techniques may be introduced into *SkyPackage* to provide ranking

REFERENCES

- [1] D. Abadi, A. Marcus, S. Madded, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. VLDB 2007.
- [2] O. Barndorff-Nielsen and M. Sobel. On the Distribution of the Number of Admissible Points in a Vector Random Sample. *Theory of Probability and its Applications*, 1966. pp. 249–269.
- [3] D. Beckett. RDF/XML Syntax Specification. Recommendation, World Wide Web Consortium, 2004. See <http://www.w3.org/TR/rdf-syntax-grammar/>.
- [4] D. Beckett and T. Berners-Lee. Turtle - Terse RDF Triple Language. World Wide Web Consortium, 2011. See <http://www.w3.org/TeamSubmission/turtle/>.
- [5] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast Linear Expected-time Algorithms for Computing Maxima and Convex Hulls. SODA 1990, pp. 179–187.
- [6] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the Average Number of Maxima in a Set of Vectors and Applications. ACM 1978. pp. 536–543.
- [7] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. IETF 1998.
- [8] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*. 2001, pp. 34-43.
- [9] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language (Second Edition). Recommendation, World Wide Web Consortium, 2010. See <http://www.w3.org/TR/xquery/>.
- [10] S. Borzsonyi, D. Kossmann, and K. Stocker. The Skyline Operator. ICDE 2001, pp. 421-430.
- [11] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. Recommendation, World Wide Web Consortium, March 2000. See <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- [12] D. Brickley and R.V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Candidate recommendation, World Wide Web Consortium, March 2000. See <http://www.w3.org/TR/2000/CR-rdf-schema-20000327>.
- [13] J. Broekstra. SeRQL: Sesame RDF query language. SWAD-Europe 2003, pp. 55-68.

- [14] J. Broekstra, A. Kampman, and F. V. Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC 2012, pp. 54-68.
- [15] J. Carrol and B. McBride. The Jena Semantic Web Toolkit. Public API, HP-Labs, Bristol, 2001. See <http://www.hpl.hp.com/semweb/jena-top.html>.
- [16] L. Chen, S. Gao, K. Anyanwu. Efficiently evaluating skyline queries on RDF databases. ESWC 2011, pp. 123-138.
- [17] J. Chomicki. Preference Formulas in Relational Queries. TODS 2003, pp. 427-466.
- [18] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. ICDE 2003, pp. 717-719.
- [19] J. Grant and D. Beckett. RDF Test Cases. Recommendation, World Wide Web Consortium, 2004. See <http://www.w3.org/TR/rdf-testcases/#ntriples>.
- [20] T. Deng, W. Fan, and F. Geerts. On the Complexity of Package Recommendation Problems. PODS 2012, pp. 261-272.
- [21] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torress Hayes. SPARQL 1.1 Protocol. Working draft, World Wide Web Consortium, September 2001. See <http://www.w3.org/TR/sparql11-protocol/>.
- [22] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. WWW 2002, pp. 592-603.
- [23] M. Khabbaz and L.V.S. Lakshmanan. TopRecs: Top-k algorithms for item-based collaborative filtering. EDBT 2011, pp. 213-224.
- [24] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski. Skyline Query Processing for Incomplete Data.
- [25] W. KieBling. Foundations of Preferences in Database Systems. VLDB 2002, pp. 311-322.
- [26] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. VLDB 2002, pp. 275-286.
- [27] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing Join and Selection Queries. VLDB 2006.
- [28] H. T. Kung, F. Luccio and F. P. Preparata. On Finding the Maxima of a Set of Vectors. JACM 1975, pp. 469-476.

- [29] O. Lassila and R. R. Swick. Resource Description Framework (RDF): Model and Syntax Specification. Recommendation, World Wide Web Consortium, February 1999. See <http://www.w3.org/TR/REC-rdf-syntax/>.
- [30] T. Berners-Lee and D. Connolly. Notation3 (N3): A readable RDF syntax. World Wide Web Consortium, 2011. See <http://www.w3.org/TeamSubmission/n3/>.
- [31] D. H. McLain. Drawing Contours from Arbitrary Data Points. *The Computer Journal*, 1974.
- [32] V. Raghavan, E. Rundensteiner. SkyDB: Skyline Aware Query Evaluation Framework. IDAR 2009.
- [33] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *TODS 2005*, pp. 41-82.
- [34] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer Verlag, 1985.
- [35] A. Seaborne. RDQL - A Query Language for RDF. World Wide Web Consortium, 2004. See <http://www.w3.org/Submission/RDQL/>.
- [36] W. Siberski, J. Z. Pan, and U. Thaden. Querying the Semantic Web with Preferences. *ISWC 2006*, pp. 612-624.
- [37] M. Sintek and S. Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. *ISWC 2002*, pp. 364-378.
- [38] A. Souzis. RxPath specification proposal. See <http://rx4rdf.liminalzone.org/RxPathSpec/>.
- [39] K.L. Tan, P.K. Eng, and B.C. Ooi. Efficient Progressive Skyline Computation. *VLDB 2001*, pp. 301-310.
- [40] M. Uschold, M. Gruninger. *Ontologies: Principles, Methods and Applications*. The Knowledge Engineering Review 1996, pp. 93-136.
- [41] A. Vlachou, C. Doukeridis, and N. Polyzotis. Skyline Query Processing Over Joins. *SIGMOD 2011*, pp. 73-84.
- [42] M. Xie, L.V.S. Lakshmanan, P. T. Wood. Breaking out of the Box of Recommendations: From Items to Packages. *RecSys 2010*, pp. 151-158.
- [43] M. Xie, L.V.S. Lakshmanan, P. T. Wood. CompRec-Trip: a Composite Recommendation System for Travel Planning. *ICDE 2011*, pp. 1352-1355.

- [44] M.L. Yiu, N. Mamoulis. Efficient Processing of Top- k Dominating Queries on Multi-Dimensional Data. VLDB 2007, pp. 483-494
- [45] Wen Jin, Martin Ester, Zengjian Hu, Jiawei Han. The Multi-Relational Skyline Operator. ICDE 2007, pp. 1276-1280.