October 2021

Enabling Declarative and Scalable Prescriptive Analytics in Relational Data

Matteo Brucato
University of Massachusetts Amherst

PACKAGE QUERIES: ENABLING DECLARATIVE AND SCALABLE PRESCRIPTIVE ANALYTICS IN RELATIONAL DATA

A Dissertation Presented

by

MATTEO BRUCATO

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2021

College of Information and Computer Sciences

PACKAGE QUERIES: ENABLING DECLARATIVE AND SCALABLE PRESCRIPTIVE ANALYTICS IN RELATIONAL DATA

A Dissertation Presented

by

MATTEO BRUCATO

Approved as to style and content by:

Peter J. Haas, Co-chair

Alexandra Meliou, Co-chair

Azza Abouzied, Member

Arya Mazumdar, Member

Senay Solak, Member

James Allan, Chair of the Faculty
College of Information and Computer Sciences

DEDICATION

To those who try with courage.

ACKNOWLEDGMENTS

Some people say their Ph.D. was the best time of their life, while some others say it was their worst. For me, it has been both. It is with great pleasure that I want to use this chance to briefly thank the people who have turned my journey here from a very tough time into the best time of my life. It was not an easy endeavor, and your efforts will always inspire me to do the same for others.

I will always be immensely grateful to Peter Haas, one of my advisors, for having taken me in what probably was the most difficult time in my Ph.D. He saw beyond the tough situation and believed in me. I would truly not be writing this thesis without his support. And collaborating with Peter has made this work something I will always be extremely proud of. I want to thank Alexandra Meliou, also one of my advisors, for having taught so much about research, for outspokenly showing me my limitations, and for maintaining her support and presence throughout the years. While Azza Abouzied was not formally one of my advisors, she has been more than that– a mentor and a true inspiration for becoming a better researcher. What I always admired the most about Azza is her ability to easily and humbly come up with groundbreaking ideas. If I will ever have an idea half as great as hers, I will consider my research career a complete success. I also want to thank my committee members: Arya Mazumdar and Senay Solak provided insightful ideas during my thesis proposal phase and important suggestions that made this thesis much stronger.

I am very grateful for the immense support that I have received from the Database research community. I want to thank my Microsoft Research mentor, Kaushik Chakrabarti,

for the opportunity to do an internship in such a great place with such great researchers. And thank you to all the amazing people I met while at MSR, especially Vivek Narasayya, Surajit Chaudhuri, Christian König, and Holger Pirk. And I want to thank Joe Hellerstein for letting me visit his lab at UC Berkeley in summer 2015.

I want to thank all the members of the DREAM Lab for making spending time in the lab so enjoyable. Anna Fariha has been an amazing collaborator, from which I learned a lot not only about research, but also about job hunting, networking, negotiations, etc.; and she has also been a great friend to me, who gave me a lot of support during tough times, including the pandemic–something I will always be grateful for. I am also thankful for the other fantastic members of the lab, including: Abhishek Roy, Yue Wang, Xiaolan Wang, Dan Zhang, Shivam Srivastava, Iro Moumoulidou, and Ryan McKenna. And I also want to thank Nishant Yadav, who has been an amazing collaborator and an important presence in a crucial time of my Ph.D.

I want to thank the amazing housemates at Gray street. For quite some time initially, it was Kyle Wray, Samer Nashed, Justin Svegliato, and I. I will always feel grateful for the way you guys have made me part of your life. You have taught me so much about your culture, and were always very eager to learn about mine. You have truly made me feel at home in America, by sharing it with me, quite literally. I loved spending time with you and finally truly understand the meaning of "hanging out". And I will always be thankful for the immense help that I received from you to grow as a person. I want to thank Shanu Vashishtha (I am still copy-pasting your last name!) for being such a great housemate and friend. Thank you for letting me teach you how to make Sicilian cookies and lasagne, and for listening to hours of me talking about my research and my job plans. And thank you for helping me pull through the first lockdown; I don't know how, but you made that crazy time actually enjoyable and full of good memories. And I also want to thank Emmanuele

Picciau for being stuck with us during that time and making it so fun. And last but not least, thank you Ameya Godbole for being such a good Rocket League student–you have talent! I also want to thank the Gray street house itself; I believe that house is a magical place and whoever goes there comes out a new, better person. And thank you, whoever bought that blue chair.

I feel so lucky for having so many amazing people around me, even though I moved to the States only recently in my life. I want to thank Nicolina Lo Russo, my "American *mamma*". You have taken me in your life as a son. I will always miss our Thanksgivings together and your stories from the past, especially the one involving Luciano Pavarotti. And I want to thank Linda Pisano, my other "American mama", for being so encouraging and positive, and such a great support for me. And thank you, Dee Waterman, for our dinners together. I will miss seeing Andrea Malaguti playing guitar with such a big smile on his face. And thank you, Anne Fancelli, for your immense kindness and generosity.

In these years, I have made so many amazing new friends. I am so grateful that I could show my beloved Sicily to my friends Dirk Ruiken and Tiffany Liu. I will miss playing board games with Pinar Ozisik and playing guitar with her boyfriend Joe. I will miss seeing Myungha Jang playing video games with quick time events. I will miss eating good Italian food with my friends Dario and Sarah Caddeo. And I will miss my time in New York City with Clemens and Christiane Rosenbaum. Thank you, Miro Mannino, for sticking around even after our time in Italy. I will miss our time in Abu Dhabi with you and Valerio Conicella. And thank you for visiting me in Amherst and for speaking Italian with me on the streets around my house; you know what incredible sequence of consequences that created in my life. And thank you, Valeria Cappelletti and Emilio Terral, for being part of those consequences. I will miss spending time with Luda Elagina and Alexey Miroshnikov speaking about math. I am glad Luis Pineda and I are still playing Rocket League together

sometimes. Thank you, Javier Burroni and Tomas Geffner for our amazing asados. Finally, I want to thank all the wonderful friends I have shared many amazing memories with: Sajia Darwish, Jennie Steshenko, Niri Karina, Francisco Garcia, Kevin Winner, Kaleigh Clary, Garrett Bernstein, Su Lin Blodgett, Yume Callahan, and Keen Sung.

Lastly, I want to thank my family. This dissertation would have not been possible without you. Thank you for your immense support that crosses oceans and continents, and is not affected by the passage of time. My parents, Andrea Brucato and Silvia Carimi, my sister, Chiara Brucato, and my brother-in-law, Fabio Tutrone, have always supported every single crazy decision I made for my career, and always believed in me. And I want to immensely thank Tamara Rossi Mercanti, my fiancée. You are the smartest, kindest, and most courageous person I know in this world. I am so grateful you are in my life. You are a constant inspiration to be the best version of me that I can. You are my lighthouse and, soon, my home. I can't wait to grow hand in hand by your side.

ABSTRACT

PACKAGE QUERIES: ENABLING DECLARATIVE AND SCALABLE PRESCRIPTIVE ANALYTICS IN RELATIONAL DATA

SEPTEMBER 2021

MATTEO BRUCATO

B.Sc., UNIVERSITY OF BOLOGNA

M.Sc., UNIVERSITY OF BOLOGNA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Peter J. Haas and Professor Alexandra Meliou

Constrained optimization problems are at the heart of significant applications in a broad range of domains, including finance, transportation, manufacturing, and healthcare. They are often found at the final step of business analytics, namely prescriptive analytics, to allow businesses to transform a rich understanding of data, typically provided by advanced predictive models, into actionable decisions. Modeling and solving these problems has relied on application-specific solutions, which are often complex, error-prone, and do not

generalize. Our goal is to create a domain-independent, declarative approach, supported and powered by the system where the data relevant to these problems typically resides: the database. Despite their widespread importance, declarative and scalable solutions to support prescriptive analytics close to the data did not exist prior to this thesis.

This thesis presents a complete system that supports package queries, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets, allowing the declarative specification and efficient evaluation of a significant class of constrained optimization problems–integer programs–within a database. Package queries pose unique challenges to a database system, ranging from their richer expressive power, more complex semantics, and harder computational complexity than their SQL counterpart, to scalability issues that arise from large amounts of data and uncertainty in the data. This thesis presents a unified system to address all these challenges. It further demonstrates the performance, quality, and applicability of our solutions with real-world problems from finance, healthcare, and science.

CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

This thesis presents the first system to support *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets, allowing the declarative specification and efficient evaluation of a significant class of constrained optimization problems–integer programs–within a relational database. Package queries are a unified solution to enable declarative and scalable prescriptive analytics close to the data.

Traditional database queries rely on a simple evaluation model: they define constraints that each record in the result must satisfy. However, many practical, real-world problems require a *collection* of result records to satisfy constraints collectively, rather than individually.

Example 1 (Meal Planner)**.** A dietitian needs to design a daily meal plan for a patient. She wants a set of three gluten-free meals, between 2,000 and 2,500 calories in total, and with a low total intake of saturated fats.

Example 2 (Night Sky)**.** An astrophysicist is looking for regions of the night sky that may contain previously unseen quasars. Regions are explored if their overall redshift is within some specified parameters, and ranked according to their likelihood of containing a quasar [83].

Similar kinds of combinatorial optimization problems arise in a variety of application domains, such as coordinating fleet and crew assignments in airline scheduling to reduce delays and costs [132], managing delinquent consumer credit to minimize losses [101], optimizing organ transplant allocation and acceptance [9], planning of cancer radiotherapy treatments [136, 150], product bundles, course selection [115], team formation [14, 94], vacation and travel planning [45, 163], and computational creativity [121]. Many of these combinatorial optimization problems can be expressed as *integer linear programs* (ILP). ILP solutions alone account for billions in US dollars of projected benefits within each of these and other industry sectors [38].

While in some of these applications the data part of the optimization is *deterministically* known, in many real-world settings, data can be uncertain (i.e., *stochastic*) at the time decisions have to be made.

Example 3 (Financial Portfolio)**.** Given uncertain predictions for future stock prices based on financial models derived from historical data, an investor wants to invest $1,000 in a set of trades (decisions on which stocks to buy and when to sell them) that will maximize the *expected future gain*, while ensuring that the *loss* (if any) will be lower than $10 with probability at least 95%.

Example 4 (Robot Path Planning)**.** Given partial and noisy knowledge of the world, provided by noisy sensors or missing information, and uncertainty in motor control, a robot needs to find the expected shortest path to reach a certain destination that also limits the probability of taking longer than a certain amount of allotted time.

Optimization and decision making under uncertainty is often found at the final step of business analytics pipelines: prescriptive analytics [17]. Despite the clear need, modeling and solving these problems has relied on application-specific solutions [14, 45, 94, 115, 121],

Recipes

	sat_fat	Kcal	gluten
t_1	7.1	450	0
t_2	5.2	550	2.5
t_3	3.2	250	0.5
t_4	6.5	150	0
t_5	2.0	1200	0.1

SQL Query

```
SELECT  *
FROM    Recipes
WHERE   gluten = 0
```

Query Result

	sat_fat	Kcal	gluten
t_1	7.1	450	0
t_4	6.5	150	0

Figure 1.1: Sample Recipes table (left), a SQL query that selects gluten-free recipes (center), and the result of the query (right). The result of a SQL query is a table containing all the tuples from the input relation that satisfy the selection predicate expressed in the WHERE **clause (e.g., gluten** $= 0$**).**

which can often be complex and error-prone, and fail to generalize. Also, while current database technology offers extensive support for all other steps of business analytics (i.e., descriptive, diagnostic and predictive analytics), there is little to no support for scalable prescriptive analytics [12, 144]. The goal of this thesis is to create a domain-independent, declarative and scalable approach for enabling prescriptive analytics, supported and powered by the system where the data relevant to these problems typically resides: the database.

1.1 Package queries for data-centric prescriptive analytics

In this thesis, we introduce a full-fledged system that supports *package queries*, a new query model that extends traditional Relational Database Management Systems (RDBMS) to handle complex constraints and preferences over answer sets.

In an RDBMS, data is stored in *relations* (or *tables*), each including a set of *tuples* (also called *rows* or *records*). The set of *attributes* (or *columns*) of a relation is called its *schema*. Figure 1.1 shows an example recipe table with schema Recipes(sat_fat, kcal, gluten). Each tuple in the table, $t_1, \ldots, t_5$, represents a recipe, with saturated fat, kilocalorie and gluten attributes. Users write their queries using a *declarative* query language, such as SQL.

A declarative query language allows users to describe the properties of all the tuples of interest, without having to worry about low-level details, such as how the data is physically stored on disk and what algorithms and data structures can be used to most efficiently retrieve the data. The goal of an RDBMS is to manage all this in a transparent way to the users. A SQL query expresses the conditions that all the result tuples have to satisfy as a Boolean *selection predicate* in the WHERE clause: a tuple either satisfies the predicate, and thus belongs to the result, or not. In the example of Figure 1.1, the SQL query selects gluten-free recipes, using the simple selection predicate $\mathsf{gluten} = 0$. The result of the query is also a relation, containing all gluten-free tuples from the input relation.

Like standard SQL queries, package queries are also defined over traditional relations. Unlike SQL, which also returns a relation as output, package queries return *packages*. A package is a collection of records that (a) individually satisfy *base predicates* (traditional selection predicates), and (b) collectively satisfy *global predicates* (package-specific predicates). Package queries are combinatorial in nature: the result of a package query is a (potentially infinite) set of packages, and an *objective criterion* can define a preference ranking among them.

Package queries encompass and extend traditional SQL queries. In the examples presented above, there are some conditions that can be verified on individual data records (e.g., gluten content, in the meal planner example). These conditions are fully supported by built-in functionalities of modern database systems. This model is computationally efficient as the database system can evaluate each record individually to determine whether it satisfies the query conditions. In this thesis, when we discuss complexity, we refer to *data complexity* [158], which, for any fixed query, expresses its complexity as a function of the size of the database instance (i.e., number of records). Traditional SQL queries have polynomial complexity.

However, there are other constraints in the examples (e.g., total calories) that can only be evaluated on a *collection* of records. For these constraints, it is never possible to tell whether a record would be allowed in a solution without knowing what other records are also part of the same solution. Traditional database technology offers little to no support for these combinatorial constraints and preferences. SQL technology has also been extended to enable more complex queries. As a result, some of these extensions (e.g., self-joins and recursion) allow SQL to express some limited package-level constraints. However, this approach has several drawbacks: it only applies to very limited types of package queries; the resulting SQL queries are very complex and hard to write; their evaluation is very inefficient with current technology. Furthermore, package queries that deal with uncertain data (like the portfolio and the path planning examples presented above), are even harder than their deterministic counterpart. Throughout this dissertation, we refer to these as *stochastic package queries*. While probabilistic databases [43, 149] offer support for SQL queries under uncertainty, they offer no support for stochastic packages.

Why support combinatorial optimization in a database system?

Extending traditional database functionality to provide support for packages, rather than supporting packages at the application level, is justified by three reasons. First, the data used to construct packages typically resides in a database system, and packages themselves are structured data objects that should naturally be stored in and manipulated by a database system. Second, having a declarative query interface allows users to reuse the same logical model of the data they are already familiar with for expressing new, more complex combinatorial problems, without having to deal with the underlining physical representation of the data. Thus tasks related to efficiently storing data, maintaining consistency, controlling access, and efficiently retrieving and preparing the data for analysis

can leverage the full power of a DBMS, while avoiding the usual slow, cumbersome, and error-prone analytics workflow where we read a dataset off of a database into main memory, feed it to stochastic-prediction and optimization packages, and store the results back into the database. Third, the features of packages and the algorithms for constructing them are not unique to each application; therefore, the burden of package support should be lifted off application developers, and database systems should support package queries like traditional queries.

1.2 Challenges

This thesis addresses five important challenges:

Declarative specification of packages. The first challenge is to support *declarative* specification of packages. SQL enables the declarative specification of properties that result tuples should independently satisfy. In Example 1, it is easy to specify the exclusion of meals with gluten using a regular SQL selection predicate. However, it is difficult to specify global constraints (e.g., total calories of a set of meals should be between 2,000 and 2,500 calories). Expressing such a query in SQL requires either complex self-joins that explode the size of the query, or recursion, which results in extremely complex queries that are hard to specify and optimize (Section 2.2). Our goal is to maintain the declarative power of SQL and its ability to express non-package constraints, while extending its expressiveness to allow for the easy specification of packages. The declarativeness of the language allows users to reuse the same logical model of the data they are already familiar with, to express combinatorial constraints and optimization objectives, without having to deal with the underlining physical representation of the data.

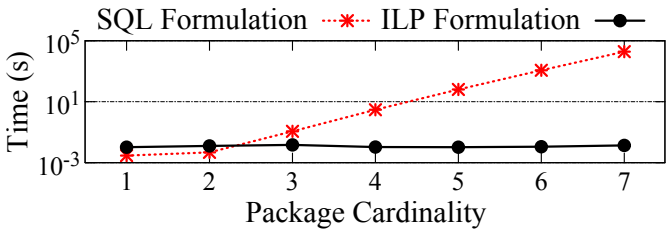


Figure 1.2: Traditional database technology is ineffective at package evaluation, and the runtime of the SQL formulation of a package query (Section 2.2) grows exponentially. In contrast, ILP solvers (Section 2.4) are more effective.

Evaluation of package queries. The second challenge pertains to the *evaluation* of package queries. Due to their combinatorial complexity, package queries are harder to evaluate than traditional database queries [50]. Package queries are in fact as hard as integer programs (Section 2.3). Existing database technology is ineffective at evaluating package queries, even if one were to express them in SQL. Figure 1.2 shows the performance of evaluating a package query expressed as a multi-way self-join query in traditional SQL (described in detail in Section 2.2) as opposed to an integer linear program (Section 2.4). As the cardinality of the package increases, so does the number of joins, and the runtime of the SQL solution quickly becomes prohibitive: in a small set of 100 tuples from the Sloan Digital Sky Survey dataset [151], SQL evaluation takes almost 24 hours to construct a package of 7 tuples. Our goal is to extend the database evaluation engine to take advantage of external tools, such as ILP solvers, which are more effective for combinatorial problems.

Evaluation of stochastic package queries. Most of the available solvers offer support for some classes of non-linear expressions, which can be used, in very limited cases, to express stochastic package queries. However, even if a stochastic package query can be expressed this way, the resulting formulation may still be too complex for solvers to optimize efficiently. A more general solution consists of using *approximate Monte*

Carlo formulations (also known as *scenario approximations* in the stochastic optimization literature [32, 35, 47, 111]). Monte Carlo databases (MCDB's) [29, 80, 90, 149] offer support for modeling data uncertainty and computing results to traditional SQL queries over stochastic relations with complex continuous distributions. Our goal is to extend MCDB's to support the construction of packages under uncertainty.

Performance and scaling to large datasets. The third challenge relates to query evaluation *performance* and *scaling* to large datasets. Integer programming solvers have two major limitations: they require the entire problem to fit in main memory, and they fail when the problem is too complex (e.g., too many variables or too many constraints). Our goal is to overcome these limitations through sophisticated evaluation methods that allow solvers to scale to large data sizes.

Performance and scaling for stochastic package queries. Monte Carlo methods usually require the generation of lots of *scenarios* (i.e., different possible realizations of the uncertain data) in order to produce feasible and close-to-optimal solutions. The number of required scenarios grows with the size of the input table and the probabilistic requirements [37, 30, 32, 35, 33]. Our goal is to overcome these limitations with sophisticated evaluation strategies to produce feasible and close-to-optimal solutions.

1.3 Contributions

This thesis presents a complete system that supports package queries, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets, allowing the declarative specification and efficient evaluation of a significant class of constrained optimization problems–integer programs–within a database. It

presents solutions to both *deterministic* and *stochastic* package queries, and shows several applications of package queries in healthcare, finance, natural language processing, and robotics, including graphical interfaces that let users specify and refine complex queries as well as navigate the solution space. Most of the algorithmic solutions presented in this thesis also include strong theoretical guarantees that allow users to obtain solutions within certain desired approximation bounds.

Declarative language, complexity and semantics

We present PaQL (Package Query Language), a declarative language that provides simple extensions to standard SQL to support constraints and objectives at the package level. We introduce sPaQL, a PaQL extension to declaratively support *expectation* and *probabilistic* global constraints and objectives over *stochastic* attributes. We prove that PaQL is at least as expressive as Integer Linear Programming (ILP), which implies that evaluation of package queries is NP-hard (Section 2.2). We provide translation rules to express any deterministic package query into an equivalent integer linear program. This translation provides *denotational semantics* of PaQL as well as a basis for our algorithmic solutions. We provide translation rules to express stochastic constraints and objectives into equivalent integer non-linear constraints. This translation provides *denotational semantics* of sPaQL as well as a basis for our algorithmic solutions for stochastic queries.

Scalable methods for deterministic package queries

We present a fundamental evaluation strategy, **Direct**, that combines the capabilities of databases and constrained-optimization solvers to derive solutions to deterministic package queries. The core of this approach is based on the translation rules that transform a package query to an integer linear program. This translation allows for the use of highly-optimized

tools for the evaluation of package queries (Section 2.4). We discuss the applicability and limitations of this approach.

We introduce an offline data partitioning strategy that allows package query evaluation to scale to large data sizes. The core of our evaluation strategy, **SketchRefine**, consists of separating the package computation into multiple stages, each with small subproblems, which the solver can evaluate efficiently. In the first stage, the algorithm "sketches" an initial sample package from a set of representative tuples, while the subsequent stages "refine" the sketched package by solving an integer program within each partition.

We prove that **SketchRefine** guarantees a $(1 \pm \epsilon)$-factor approximation compared to **Direct**, where ϵ is a flexible parameter of the offline partitioning. (Section 3.2).

We present an extensive experimental evaluation on both real-world data and the TPC-H benchmark (Section 5.2.2) that shows that our query evaluation method **SketchRefine**: (1) is able to produce packages an order of magnitude faster than the integer solver used directly on the entire problem; (2) scales up to sizes that the solver cannot manage directly; (3) produces packages of very good quality in terms of objective value; (4) is robust to partitioning built in anticipation of different workloads.

We design a parallel version of **SketchRefine** that can efficiently solve queries that require most of the partitions to be accessed. We experimentally show that this type of query is a worst case for the offline data partitioning used by **SketchRefine**, and severely impacts the sequential performance of the algorithm.

We present an empirical study on *preconditioning* solvers with starting solutions. Our results show that seeding solvers with feasible packages can significantly improve the performance of the solver especially on harder queries.

Scalable methods for stochastic package queries

We extend **Direct** to support exact translation of expectation and probabilistic constraints into deterministic constraints that off-the-shelf optimizers can solve, and discuss the challenges that these formulations pose. The complexity of probabilistic constraints only makes this translation possible for stochastic attributes that follow a Gaussian distribution with known mean and variance.

We present **Naïve**, an extension to **Direct** to support expectation and probabilistic constraints for any type of stochastic attribute distribution, including cases where the distribution is unknown and only generators are available. The core of this approach is based on translation rules that transform a stochastic package query into an integer program with indicator constraints, each of which governs the feasibility of a single scenario generated with Monte Carlo simulations. We discuss the applicability and limitations of this approach.

We introduce the concept of a *summary of scenarios*, and show that summaries can replace real scenarios in **Naïve** to produce feasible solutions much more efficiently, by largely increasing the number of Monte Carlo scenarios that can be included without paying the cost of optimizing large, complex integer programs. The core of our evaluation strategy, **SummarySearch**, consists of separating the package computation into multiple iterations, each with a small set of summaries, rather than lots of real scenarios, which the solver can evaluate efficiently.

We present an extensive experimental evaluation on real-world data that shows that: (1) **SummarySearch** is always able to find feasible solutions, while **Naïve** cannot in most cases—when both **SummarySearch** and **Naïve** can find feasible solutions, **SummarySearch** is often faster by orders of magnitude; (2) The packages produced by **SummarySearch** are of high quality (low empirical approximation ratio), sometimes even

better than **Naïve** when they both produce feasible solutions; (3) Increasing the number of optimization scenarios helps **SummarySearch** find feasible solutions, and the number of optimization scenarios required by **SummarySearch** to start producing feasible solutions is much smaller than **Naïve**, explaining the orders of magnitude improvement in running time; (4) Increasing the number of summaries helps **SummarySearch** find higher-quality solutions; (5) A larger number of input tuples negatively impacts the running time of both algorithms, but **SummarySearch** is still orders of magnitude faster than **Naïve**, and finds feasible solutions with better empirical approximation ratios than **Naïve**.

1.4 Outline

Chapter 2 introduces background terminology and semantics for packages, presents our declarative language for expressing package queries, and analyzes its semantics, complexity and expressiveness.

Chapter 3 introduces our evaluation methods for package queries on *deterministic* data. We first develop a **Direct** translation of package queries into integer linear programs (ILP), a basic evaluation method more suitable for small datasets. Then, we extend this technique to our main algorithm, **SketchRefine**, which supports efficient package evaluation for large datasets. The chapter concludes with an interface for specifying and manipulating package queries, demonstrated on the Meal Planner application.

Chapter 4 provides methods for processing *stochastic package queries* (SPQs) in order to solve optimization problems over uncertain data. We first introduce a **Naïve** algorithm based on prior work in stochastic programming [139], able to solve SPQs that do not require many scenarios. Then, we provide a novel **SummarySearch** algorithm that, instead of trying to solve a large deterministic problem like **Naïve** does, seamlessly approximates it

via a sequence of smaller problems defined over carefully crafted *summaries* of the scenarios that accelerate convergence to a feasible and near-optimal solution. The chapter concludes with presenting sPaQLTooLs, an interface that allows decision makers to quickly identify optimal choices despite the uncertainty and size of the data; sPaQLTooLs is demonstrated using the Financial Portfolio application.

Chapter 5 identifies the major limitations of the solutions presented in this thesis and future research to address them. Further, the chapter presents several important future research directions connected to package queries: querying package specification by example (PQBE) to help non-expert users, and incremental package evaluation (IPE) that has the potential to improve the performance of package query evaluation even further. The chapter includes preliminary results to help delineate the major research challenges of PQBE and IPE.

Chapter 6 describes the related research in depth. Being highly interdisciplinary, package queries connect several areas of Computer Science, such as Database Systems, Approximation Theory, and Query Languages, and areas of Operations Research, such as Integer and Stochastic Programming, Constrained Optimization, as well as areas at the intersection between the two, such as Business Analytics, Planning, Robotics, and Artificial Intelligence.

Chapter 7 concludes this thesis with a summary of the contributions and closing remarks.

CHAPTER 2

LANGUAGE, SEMANTICS AND COMPLEXITY OF PACKAGE QUERIES

In this chapter, we first introduce background terminology and semantics for packages. We then present our declarative language for expressing package queries, and analyze its semantics, complexity and expressiveness.

2.1 Background

We first introduce some basic notation and describe the semantics of packages.

2.1.1 Tuple, relation and package semantics

Given sets A_1, ..., A_k, a *tuple* (or *record*), denoted by $(a_1, \ldots, a_k)$, is an ordered sequence of elements from each set, that is, $a_1 \in A_1$, ..., $a_k \in A_k$. We call sets A_1, ..., A_k the *attributes* of a tuple, and each value $a_1, \ldots, a_k$ a tuple value or *entry*.

Let U be the *universe* of possible *tuples* of a relation R, and $\mathbb{N}$ the set of all the natural numbers, $\mathbb{N} = \{0, 1, \ldots\}$. R is a multiset over universe U, denoted as (U, m_{R}), where $m_{\mathsf{R}} : U \to \mathbb{N}$ is a multiplicity function, indicating the number of occurrences of each element of U in R. We also call the set of attributes A_1, ..., A_k the *schema* of relation R, denoted by $\mathsf{R}(A_1, \ldots, A_k)$. A *database* is a set of relations, and a *workload* a set of queries over the database.

A *package* P_R, defined over R, is a multiset with multiplicity $m_{\mathsf{P}_\mathsf{R}} : U \to \mathbb{N}$, such that $\forall t \in U : m_\mathsf{R}(t) = 0 \implies m_{\mathsf{P}_\mathsf{R}}(t) = 0$. In other words, a package can only include tuples from its relation, but it may do so with arbitrary multiplicity. The schema of a package is the same as its defining relation. The goal of a *package query* is to identify the multiplicities of its resulting package.

Throughout the thesis, we use the following multiset operators. Given relations (or packages) R_1 and R_2, we say that $\mathsf{R}_1 \subseteq \mathsf{R}_2$ iff $\forall t \in U : m_{\mathsf{R}_1}(t) \leq m_{\mathsf{R}_2}(t)$; $\mathsf{R}_1 \cup \mathsf{R}_2$ has multiplicity $m_{\mathsf{R}_1 \cup \mathsf{R}_2}(t) = m_{\mathsf{R}_1}(t) + m_{\mathsf{R}_2}(t)$, $\forall t \in U$; $\mathsf{R}_1 \setminus \mathsf{R}_2$ has multiplicity $m_{\mathsf{R}_1 \setminus \mathsf{R}_2}(t) = \max\{0, m_{\mathsf{R}_1}(t) - m_{\mathsf{R}_2}(t)\}$, $\forall t \in U$.

2.1.2 Monte Carlo relations and stochastic package queries

To offer support for stochastic package queries with data following complex continuous (or discrete) distributions (see Examples 8 and 4 in Chapter 1), we employ the Monte Carlo data model [149], first introduced in the Monte Carlo database, MCDB [80, 79]. In a Monte Carlo relation, a tuple entry can be a *random variable*, rather than a deterministic value. A Monte Carlo relation stores a *variable generation function* (VG function) for each random variable, which can be used by the database engine to sample values for that variable. Multiple variables can have their values generated by the same VG function in order to capture statistical dependencies. This approach encompasses the traditional *attribute-level*, discrete, probabilistic data models [149], which explicitly store all the discrete values and associated probabilities each random variable can take. The semantics of a relation is the the so-called *possible worlds semantics*: a possible world is one in which all random variables have been sampled.

If there are no random variables, or all possible worlds are the same, we say that the relation is *deterministic*, otherwise *stochastic*. We call a package query defined over a

stochastic relation a *stochastic package query*. The result of a stochastic package query is always deterministic: all multiplicities of the resulting package are deterministically chosen, despite the stochastic nature of the input relation.

2.2 Declarative language support for packages

Database systems do not natively support package queries. While there are ways to express *some* package queries in SQL, these are cumbersome and inefficient. In this section, we first describe two ways of expressing some deterministic types of package queries in SQL and explain their limitations and drawbacks. We then describe PaQL, a declarative query language for specifying packages, and sPaQL, an extension to PaQL to support stochastic package queries. Throughout this thesis, we always refer to PaQL for deterministic package queries and sPaQL for stochastic package queries. Unless otherwise noted, a "package query" is always deterministic.

2.2.1 Expressing package queries with SQL

Specifying packages with self-joins

In the limited case of packages over deterministic relations and with strict cardinality, i.e., a fixed number of tuples, it is possible to express package queries using relational self-joins. The query of Example 1 requires three meals (a package with cardinality three), and can be expressed as a three-way self-join:

```
SELECT     *
FROM       Recipes R1, Recipes R2, Recipes R3
WHERE      R1.pk < R2.pk AND R2.pk < R3.pk AND
           R1.gluten = 0 AND R2.gluten = 0 AND R3.gluten = 0 AND
           R1.kcal + R2.kcal + R3.kcal BETWEEN 2.0 AND 2.5
ORDER BY   R1.sat_fat + R2.sat_fat + R3.sat_fat
```

PaQL syntax specification

```
SELECT PACKAGE(∗|column_name [,...])
               [AS] package_name
FROM relation_name [AS] relation_alias
               [REPEAT repeat] [,...]
[ WHERE w_expression ]
[ SUCH THAT st_expression ]

[ (MINIMIZE|MAXIMIZE) obj_expression ]
```

PaQL query for Example 1

```
Q: SELECT     PACKAGE(∗) AS P
   FROM       Recipes R REPEAT 0
   WHERE      R.gluten = 0
   SUCH THAT
      COUNT(P.∗) = 3 AND
      SUM(P.kcal) BETWEEN 2.0 AND 2.5
   MINIMIZE   SUM(P.sat_fat)
```

Figure 2.1: Specification of the PaQL syntax (left), and the PaQL query for the MEAL PLANNER query from Example 1 (right).

Such a query is efficient only for constructing packages with very small cardinality: larger cardinality requires a larger number of self-joins, quickly rendering evaluation time prohibitive (Figure 1.2). The benefit of this specification is that the optimizer can use the traditional relational algebra operators, and augment its decisions with package-specific strategies. However, this method does not apply for packages of unbounded cardinality.

Specifying packages using recursion

SQL can express package queries by generating and testing each possible subset of the input relation. This requires recursion to build a *powerset table*; checking each set in the powerset table for the query conditions will yield the result packages. This approach has three major drawbacks. First, it is not declarative, and the specification is tedious and complex. Second, it is not amenable to optimization in existing systems. Third, it is extremely inefficient to evaluate, because the powerset table generates an exponential number of candidates.

2.2.2 PaQL: the Package Query Language

Our goal is to support declarative and intuitive package specification. In this section, we describe PaQL, a declarative query language that introduces simple extensions to SQL to define package semantics and package-level constraints.

PaQL syntax

Figure 2.1 shows the general syntax of PaQL (left) and the specification for the query of Example 1 (right), which we use as a running example to demonstrate PaQL's features. Square brackets enclose optional clauses and arguments, and a vertical bar separates syntax alternatives. In this specification, **`repeat`** is a non-negative integer; **`w_expression`** is a Boolean expression over tuple values (as in standard SQL), and can only contain references to **`relation_name`** and **`relation_alias`**; **`st_expression`** is a Boolean expression and **`obj_expression`** is an expression over aggregate functions or SQL subqueries with aggregate functions; both **`st_expression`** and **`obj_expression`** can only contain references to **`package_name`**, which specifies the name of the package result.

Basic package query

The new keyword PACKAGE differentiates PaQL from traditional SQL queries.

$\mathcal{Q}_1$: SELECT $*$ FROM Recipes R

$\mathcal{Q}_2$: SELECT **PACKAGE**$(*)$ FROM Recipes R

The semantics of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are fundamentally different: $\mathcal{Q}_1$ is a traditional SQL query, with a unique, finite result set (the entire Recipes table), whereas there are infinitely many packages that satisfy the package query $\mathcal{Q}_2$: all possible *multisets* of tuples from the input relation. Each tuple, whether or not unique in the input relation, has unbounded multiplicity in the package. The result of a package query like $\mathcal{Q}_2$ is a set of packages.

Each package resembles a relational table containing a collection of tuples (with possible repetitions) from the relation Recipes. A package result of $\mathcal{Q}_2$ follows the schema of Recipes. Similar to SQL, the PaQL syntax allows the specification of the output schema in the SELECT clause. For example, PACKAGE(sat_fat, kcal) only returns the saturated fat and calorie attributes of the package.[1]

The language also permits multiple relations in the FROM clause; in that case, the packages produced will follow the schema of the join result. In the remainder of this thesis, we focus on package queries without joins. This is for two reasons: (1) The join operation is part of traditional SQL and can occur before package-specific computations. (2) There are important implications in the consideration of joins that extend beyond the scope of our work. Specifically, materializing the join result is not always necessary, but rather, there are space-time trade-offs and system-level solutions that can improve query performance in the presence of joins. These extensions are orthogonal to the techniques we present in this work.

Although semantically valid, a query like $\mathcal{Q}_2$ would not occur in practice, as most application scenarios expect few, or even exactly one result. We proceed to describe the additional constraints in the example query $\mathcal{Q}$ (Figure 2.1) that restrict the number of package results.

Repetition constraints

The REPEAT 0 statement in query $\mathcal{Q}$ from Figure 2.1 specifies that each tuple from the input relation Recipe can appear in a package result at most once (no repetitions are allowed). If a tuple has duplicates in the input table (multiplicity greater than 1),

[1]This syntax slightly differs from the one presented in [25].

repetition restrictions are applied on each individual duplicate. Formally, for input table R, REPEAT ρ, $\rho \geq 0$, implies $\forall t \in U : m_{\mathsf{P}}(t) \leq m_{\mathsf{R}}(t)(1+\rho)$. If this restriction is absent (as in query $\mathcal{Q}_2$), the multiplicity of a tuple is unbounded. By allowing no repetitions, $\mathcal{Q}$ restricts the package space from infinite to 2^n, where n is the size of the input relation. Generalizing, REPEAT ρ allows a package to repeat tuples up to ρ times, resulting in $(2+\rho)^n$ candidate packages. Tuple repetitions naturally appear in many problems (e.g., Example 8, where multiple shares of the same investment asset can be included in a portfolio). While the PaQL specification allows for an arbitrarily large number of repetitions, we expect that systems will impose a default bound in practice. In this thesis, we focus on queries with explicit repetition constraints.

Base and global predicates

A package query defines two types of predicates. A *base predicate*, defined in the WHERE clause, is equivalent to a selection predicate and can be evaluated with standard SQL: any tuple in the package needs to *individually* satisfy the base predicate. For example, query $\mathcal{Q}$ from Figure 2.1 specifies the base predicate: R.gluten $= 0$. Since base predicates directly filter input tuples, they are specified over the input relation R. *Global predicates* are the core of package queries, and they appear in the new SUCH THAT clause. Global predicates are higher-order than base predicates: they cannot be evaluated on individual tuples, but on tuple collections. Since they describe package-level constraints, they are specified over the package result P, e.g., COUNT(P.$*$) $= 3$, which limits the query results to packages of exactly 3 tuples.

The global predicates in query $\mathcal{Q}$ abbreviate aggregates that are in reality SQL subqueries. For example, COUNT(P.$*$)=3, abbreviates (SELECT COUNT($*$) FROM P)=3. Using

subqueries, PaQL can express arbitrarily complex global constraints among aggregates over a package.

Objective clause

The objective clause specifies a ranking among candidate package results, and appears with either the MINIMIZE or MAXIMIZE keyword. It is a condition on the package-level, and hence it is specified over the package result P, e.g., MINIMIZE SUM(P.sat_fat). Similar to global predicates, this form is a shorthand for MINIMIZE (SELECT SUM(sat_fat) FROM P). A PaQL query with an objective clause returns a single result: a package that optimizes the value of the objective. Note that if there is more than one optimal package (i.e., if it is possible to construct more packages with the same optimal objective value) the specific optimal package that is returned only depends on the solution algorithm (e.g., starting conditions) and is beyond the control of the user. The evaluation methods that we present in this work focus on queries with an objective clause.

While PaQL allows arbitrary aggregate functions in the global predicates and the objective clause, in this work, we only support package queries with *linear* aggregates over numerical variables. A linear aggregate can be a constant or an attribute value multiplied by a constant, or any linear combination thereof. We defer the study of non-linear aggregates and UDFs to future work.

sPaQL: Expectation and probabilistic predicates and objectives

We further extend the PaQL syntax to support two types of stochastic global predicates: expectation and probabilistic predicates. The following query shows the sPaQL specification for the stochastic package query of Example 8.

```
SELECT    PACKAGE(*) AS Portfolio
```

FROM Stock_Investments
SUCH THAT
SUM(price) ≤ 1000 AND
SUM(Gain) ≥ -10 **WITH PROBABILITY** ≥ 0.95
MAXIMIZE **EXPECTED** SUM(Gain)

Throughout the dissertation, we denote stochastic attributes with a capital letter (e.g., Gain) to differentiate them from deterministic attributes (e.g., price). The values of Gain are, in fact, unknown at query time. The table Stock_Investments only stores a generator function (e.g., a UDF) for each tuple under attribute Gain. If a stochastic attribute appears in a predicate (or objective), as in this example, the system does not allow it to be deterministic. For example, SUM(Gain) ≥ -100, without WITH PROBABILITY ≥ 0.95, is not allowed. The system supports two types of stochastic predicates and objectives: *expectation* and *probabilistic* predicates.

An *expectation global predicate* can be formed by prepending the keyword EXPECTED to a global predicate that involves at least one stochastic attribute. Similarly, we can define an *expectation objective*, as in the query example above. In this example, we call what comes after EXPECTED, i.e. SUM(Gain), the *deterministic part* of the objective clause.

A probabilistic global predicate can be formed by appending the expression WITH PROBABILITY $\geq p$ (or $\leq p$) to a global predicate that involves at least one stochastic attribute, as shown in the query above, where $0 \leq p \leq 1$. In this example, SUM(Gain) ≥ -10 is the deterministic part of the global predicate.

Stochastic package queries that only include expectations are linear, thanks to the linearity of expectations. However, queries that include probabilistic global predicates or objectives are usually non-linear, due to the complexity of the probabilities, as we discuss in more detail in later sections. Nonetheless, probabilistic predicates having linear deterministic parts offer tractable solutions [26, 28]. In this thesis, we restrict our focus to probabilistic constraints with a linear deterministic part.

Figure 2.2 shows the syntax diagram for sPaQL, constructed using the Railroad Diagram Generator [127]. In PaQL, a linear constraint has the general form:

(SELECT SUM(f(R)) WHERE <selection-predicate> FROM P) $\geq v$

where P is a reference name (alias) to the result package, $f(\mathsf{R})$ a function of the attributes of R (e.g., $f(\mathsf{R}) = 3\mathsf{A}_1^2 - 2\sqrt{\mathsf{A}_2} + 1$ for a table R with two attributes A_1 and A_2), and $v \in \mathbb{R}$. Syntactic sugar for a simple single-attribute, no-selection constraint is $\mathsf{SUM}(\mathsf{A}) \geq v$, where $f(\mathsf{R}) = \mathsf{A}$, for some attribute A. For example, $\mathsf{SUM}(\textbf{price}) \leq 1000$ from the query in the introduction is a single-attribute summation constraint on **price**. A cardinality constraint is a special case of a summation constraint, $\mathsf{COUNT}(*) = \mathsf{SUM}(1)$, where $f(\mathsf{R}) = 1$.

If any of the attributes in $f(\mathsf{R})$ are stochastic, sPaQL allows users to write either an expected or a probabilistic version of the constraint. An expected constraint simply prepends the keyword EXPECTED to a deterministic constraint, e.g., EXPECTED $\mathsf{SUM}(\mathsf{A}) \geq v$. Similarly, an expected minimization objective can be expressed as MINIMIZE EXPECTED $\mathsf{SUM}(\mathsf{A})$. For example, the objective function of query $\mathcal{Q}$ from the introduction maximizes the EXPECTED $\mathsf{SUM}(\mathsf{Gain})$.

A probabilistic constraint can be expressed by appending WITH PROBABILITY $\geq p$ to a deterministic constraint, for some $p \in (0,1)$. For example, $\mathsf{SUM}(\mathsf{Gain}) \geq -10$ WITH PROBABILITY ≥ 0.95. The language also allows for opposite constraints ($\leq p$) for convenience, but they can always be equivalently rewritten in the other form by flipping the inequality sign of the inner constraint and using $1-p$ instead. A probabilistic objective is expressed by prepending PROBABILITY OF to a constraint, e.g., MAXIMIZE PROBABILITY OF $\mathsf{SUM}(\mathsf{Gain}) \geq -10$.

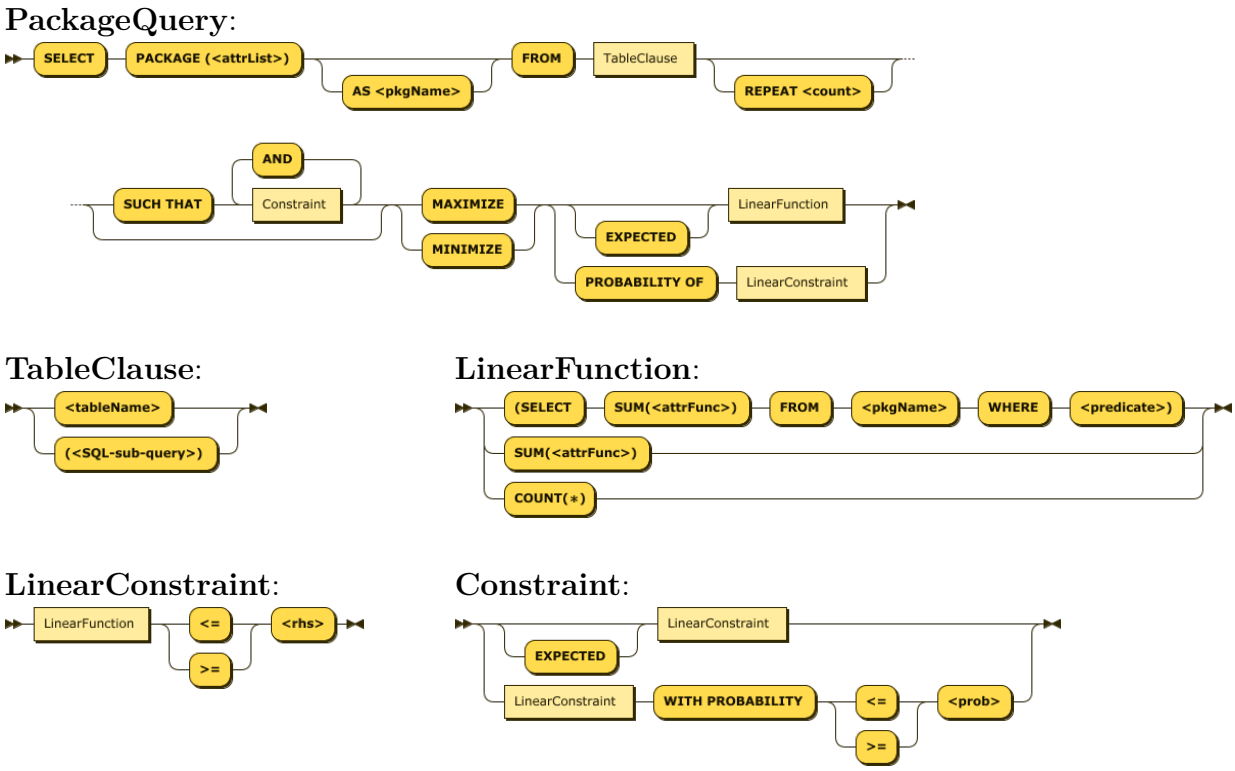


Figure 2.2: Syntax (railroad) diagram of sPaQL.

2.3 Expressiveness and complexity of PaQL

Package queries can model a great variety of problems. They are at least as expressive as integer linear programs (ILP), and, therefore, at least as hard.

Theorem 1 (Expressiveness of PaQL)**.** *Every integer linear program can be expressed as a package query in* PaQL*.*

Proof. We prove the result through a reduction from an ILP problem to a PaQL query. The reduction involves two mappings: (1) a mapping from a general ILP instance $\mathcal{I}$ to a PaQL query $\mathcal{Q}_\mathcal{I}$; (2) a mapping from a solution to the ILP problem to a package p. The mappings are such that the solution to the ILP is an optimal solution to $\mathcal{I}$ *iff* p is an optimal package for $\mathcal{Q}_\mathcal{I}$. Let $\mathcal{I}$ be an ILP problem involving n integer variables,[2] k linear constraints, and real coefficients a_i, b_{ij} and c_j:

$$\begin{aligned} \mathcal{I}: \quad \max \quad & \textstyle\sum_{i=1}^{n} a_i x_i \\ \text{s.t} \quad & \textstyle\sum_{i=1}^{n} b_{ij} x_i \leq c_j \quad \forall j = 1, \ldots, k \\ & x_i \geq 0, x_i \in \mathbb{Z} \quad \forall i = 1, \ldots, n \end{aligned}$$

The PaQL query $\mathcal{Q}_\mathcal{I}$ constructed from $\mathcal{I}$ is:

$\mathcal{Q}_\mathcal{I}$: SELECT PACKAGE($*$) AS P FROM (
SELECT a_1 AS attr$_{obj}$, b_{11} AS attr$_1$, …, b_{1k} AS attr$_k$
UNION $\cdots$
SELECT a_n AS attr$_{obj}$, b_{n1} AS attr$_1$, …, b_{nk} AS attr$_k$)
SUCH THAT SUM(P.attr$_1$) $\leq c_1$ AND … SUM(P.attr$_k$) $\leq c_k$
MAXIMIZE SUM(P.attr$_{obj}$)

Let $\hat{x}$ be an *assignment* to the variables in $\mathcal{I}$. Package p is constructed from $\hat{x}$ by including tuple t_i exactly $\hat{x}_i$ times.

[2] For ease of presentation, we show an ILP with nonnegative variables, but the mapping generalizes to arbitrary integer variables: negative variables negate the corresponding values in the query; for arbitrary bounds on each variable, add cardinality constraints to individual tuples.

($\Rightarrow$) Suppose $\hat{x}$ is an optimal feasible solution to $\mathcal{I}$. Then $\forall j = 1, \ldots, k, \sum_{i=1}^{n} b_{ij}\hat{x}_i \leq c_j$ and $\sum_{i=1}^{n} a_i\hat{x}_i$ is maximal. Thus, by construction of p, $\forall j = 1, \ldots, k$, $\mathsf{SUM}(p.\mathsf{attr}_j) = \sum_{i=1}^{n} b_{ij}\hat{x}_i \leq c_j$, and $\mathsf{SUM}(p.\mathsf{attr}_{obj}) = \sum_{i=1}^{n} a_i\hat{x}_i$ is maximal. Therefore, p is an optimal package for query $\mathcal{Q}_{\mathcal{I}}$.

($\Leftarrow$) If p is an optimal package for $\mathcal{Q}_{\mathcal{I}}$, then, by definition, $\forall j = 1, \ldots, k, \sum_{i=1}^{n} b_{ij}\hat{x}_i \leq c_j$ and $\sum_{i=1}^{n} a_i\hat{x}_i$ is maximal. ■

As a direct consequence of Theorem 1, we obtain the following result on the complexity of package query evaluation.

Corollary 1 (Complexity of Package Queries)**.** *Package queries are NP-hard.*

The corollary follows from the fact that ILPs are NP-hard [137]. In Section 2.4, we extend the result of Theorem 1 to also show that every PaQL query over any database instance can be encoded as an integer linear program, through a set of translation rules.

2.4 Integer programming semantics of package queries

In this section, we present an ILP formulation for package queries. This formulation is at the core of our evaluation methods **Direct** and **SketchRefine**. The results presented in this section are inspired by the translation rules employed by Tiresias [104] to answer *how-to queries*. However, there are several important differences between how-to and package queries, which we extensively discuss in the overview of the related work (Chapter 6). A *translation* is a procedure that transforms a PaQL (or sPaQL) query into an *equivalent* integer program thatis readily solvable by an existing solver. Because solvers typically employ heuristics and approximations, the actual result produced by the solver may, in

practice, be not exact. Our discussion abstracts from solver-specific features and only concentrates on the exactness of the problem formulation given to the solver.

2.4.1 PaQL to ILP translation

Let R indicate the input relation of the package query, $n = |\mathsf{R}|$ be the number of tuples in R, $\mathsf{R.attr}$ an attribute of R, P a package, f a linear aggregate function (such as COUNT and SUM), $\odot \in \{\leq, \geq\}$ a constraint inequality, and $v \in \mathbb{R}$ a constant. For each tuple t_i from R, $1 \leq i \leq n$, the ILP problem includes a nonnegative integer variable x_i, $x_i \geq 0$, indicating the number of times t_i is included in an answer package. We also use $\bar{x} = \langle x_1, x_2, \ldots, x_n \rangle$ to denote the vector of all integer variables. A PaQL query is formulated as an ILP problem using the following translation rules.

Repetition constraint. The REPEAT keyword, expressible in the FROM clause, restricts the domain that the variables can take on. Specifically, $\mathsf{REPEAT}\ \rho$ implies $0 \leq x_i \leq \rho + 1$.

Base predicate. Let β be a base predicate, e.g., $\mathsf{R.gluten} = 0$, and R_β the relation containing tuples from R satisfying β. We encode β by setting $x_i = 0$ for every tuple $t_i \notin \mathsf{R}_\beta$.

Global predicate. Each global predicate in the $\mathsf{SUCH\ THAT}$ clause takes the form $f(\mathsf{P}) \odot v$. For each such predicate, we derive a linear function $f'(\bar{x})$ over the integer variables. A cardinality constraint $f(\mathsf{P}) = \mathsf{COUNT}(\mathsf{P}.*)$ is translated into a linear function $f'(\bar{x}) = \sum_i x_i$. A summation constraint $f(\mathsf{P}) = \mathsf{SUM}(\mathsf{P.attr})$ is translated into a linear function $f'(\bar{x}) = \sum_i (t_i.\mathsf{attr}) x_i$. We further illustrate the translation with two non-trivial examples:

- $\mathsf{AVG}(\mathsf{P.attr}) \leq v$ is translated as

$$\textstyle\sum_i (t_i.\mathsf{attr}) x_i / \sum_i x_i \leq v \;\equiv\; \sum_i (t_i.\mathsf{attr} - v) x_i \leq 0$$

- (SELECT COUNT$(*)$ FROM P WHERE P.carbs > 0) $\geq$ (SELECT COUNT$(*)$ FROM P WHERE P.protein ≤ 5) is translated as

$$\textstyle\sum_i (\mathbb{1}_{\mathsf{R_c}}(t_i) - \mathbb{1}_{\mathsf{R_p}}(t_i)) x_i \geq 0$$

where

$$\mathsf{R_c} := \{t_i \in \mathsf{R} \mid t_i.\mathsf{carbs} > 0\}$$
$$\mathsf{R_p} := \{t_i \in \mathsf{R} \mid t_i.\mathsf{protein} \leq 5\}$$
$$\mathbb{1}_{\mathsf{R_c}}(t_i) := 1 \text{ if } t_i \in \mathsf{R_c};\ 0 \text{ otherwise}$$
$$\mathbb{1}_{\mathsf{R_p}}(t_i) := 1 \text{ if } t_i \in \mathsf{R_p};\ 0 \text{ otherwise.}$$

General Boolean expressions over the global predicates can be encoded into a linear program using Boolean variables and linear transformation tricks found in the literature [19].

Objective clause. We encode MAXIMIZE $f(\mathsf{P})$ as $\max f'(\bar{x})$, where $f'(\bar{x})$ is the encoding of $f(\mathsf{P})$. Similarly MINIMIZE $f(\mathsf{P})$ is encoded as $\min f'(\bar{x})$.

We call the relations R_β, R_c, and R_p described above *base relations*.

Example 5 (ILP Translation)**.** Figure 2.3 shows a toy example of the Recipes table, with two columns and 5 tuples. To transform $\mathcal{Q}$ into an ILP, we first create a non-negative, integer variable for each tuple: $x_1, \ldots, x_5$. The cardinality constraint specifies that the sum of the x_i variables should be exactly 3. The global constraint on SUM(P.kcal) is

	Recipes		
	sat_fat	Kcal	
t_1	7.1	450	$x_1 = 0$
t_2	5.2	550	$x_2 = 1$
t_3	3.2	250	$x_3 = 1$
t_4	6.5	150	$x_4 = 0$
t_5	2.0	1200	$x_5 = 1$

$$
\begin{aligned}
\min \quad & 7.1x_1 + 5.2x_2 + 3.2x_3 + 6.5x_4 + 2.0x_5 \\
\text{s.t.} \quad & x_1 + x_2 + x_3 + x_4 + x_5 = 3 \\
& 450x_1 + 550x_2 + 250x_3 \\
& \qquad + 150x_4 + 1200x_5 \geq 2000 \\
& 450x_1 + 550x_2 + 250x_3 \\
& \qquad + 150x_4 + 1200x_5 \leq 2500 \\
& x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}
\end{aligned}
$$

Figure 2.3: Example ILP formulation and solution for query $\mathcal{Q}$, on a sample Recipes dataset. There are only two packages that satisfy all the constraints, namely $\{t_2, t_3, t_5\}$ and $\{t_1, t_2, t_5\}$, but the first one is the optimal because it minimizes the objective function.

formed by multiplying each x_i with the value of the kcal column of the corresponding tuple, and specifying that the sum should be between 2 and 2.5. The objective of minimizing SUM(P.sat_fat) is similarly formed by multiplying each x_i with the sat_fat value of the corresponding tuple.

This formulation, together with Theorem 1, shows that package queries correspond *exactly* to ILP problems. This transformation is at the core of our package evaluation methods, **Direct**, which employs it directly to generate solutions for a PaQL query (Section 3.1). However, due to the limitations of ILP solvers, it is not efficient or scalable in practice. To make package evaluation practical, we develop **SketchRefine** (Section 3.2), a technique that augments the ILP transformation with a partitioning mechanism, allowing package evaluation to scale to large datasets. In Section 3.5, we show how to parallelize **SketchRefine**, in order to efficiently answer queries that require most of the partitions to be accessed.

2.4.2 sPaQL to IP translation

An exact translation from sPaQL to an integer (possibly quadratic) program is only possible under these conditions:

- For any expectation predicate (or objective clause) in the query, the expected value of each tuple's attribute involved is known exactly or can be analytically derived. For example, all tuples follow Gaussian or Poisson distributions with known (and possibily different) means. For simplicity of exposition, let us assume the expected value of a stochastic attribute Attr is stored as a deterministic attribute named exp_Attr.
- Any probabilistic predicate in the query only involves tuples following Gaussian distributions with known expectations and variances, and if the random variables have correlation, their covariance is also known exactly.[3] The resulting constraint must also be convex. A sufficient condition for convexity is that, in every probabilistic constraint, p is greater than or equal to 0.5. In this case, we also assume that the standard deviation of Attr is stored as a deterministic attribute std_Attr.

If no probabilistic predicate is present, the resulting integer program is also linear, otherwise quadratic. Notice that, in this translation, we do not use the generator functions to generate samples of the stochastic attributes. We directly use the parameters (e.g., mean and variance) of the distribution of each tuple. If these parameters are not known, or not analytically derivable, this method cannot be applied. We discuss in later chapters how Monte Carlo simulations can be used in all cases in which this direct formulation is not possible to generate *approximate* formulations.

---

[3]Notice that other distributions, besides Gaussian, might also be admissible for this approach, as long as their sum is analytically computable. However, to the best of our knowledge, no one has shown a translation of probabilistic constraints into integer programs other than for variables following Gaussian distributions.

Expectation predicate. An expectation global predicate takes the form $\mathbb{E}(f(\mathsf{P})) \odot v$ where, like before, f is a linear aggregate function (such as COUNT and SUM). Thanks to the linearity of expectations, $\mathbb{E}(f(\mathsf{P})) = f(\mathbb{E}(\mathsf{P}))$. For example, $\mathbb{E}(\mathsf{SUM}(\mathsf{P.Attr})) = \mathsf{SUM}(\mathbb{E}(\mathsf{P.Attr})) = \mathsf{SUM}(\mathsf{P.exp\_Attr})$, and $\mathbb{E}(\mathsf{COUNT}(*)) = \mathsf{COUNT}(\mathbb{E}(*)) = \mathsf{COUNT}(*)$. Therefore, expectation predicates are simply replaced with their analogous deterministic predicate over the expected values of tuples. These are translated following the same rules for PaQL (Section 2.4.1).

Probabilistic predicate. Consider a probabilistic predicate of the form

$$\Pr(f(\mathsf{P.Attr}) \geq v) \geq p,$$

where $v \in \mathbb{R}$, $p \in (0.5, 1]$ and f is a linear aggregate function (such as COUNT and SUM). It is a known fact that such a constraint is convex [139]. To translate this predicate we first introduce a new *continuous* variable c ($c \in \mathbb{R}$) into the integer program, and add a new linear constraint:

$$z_p c \leq \sum_{i=1}^{n} t_i.\mathsf{exp\_Attr} * x_i - v \tag{2.1}$$

where $z_p = \Phi^{-1}(p) \in \mathbb{R}$ is the p-quantile of the standard Gaussian distribution. If the distributions of tuples are all independent, we also add a quadratic constraint:

$$c^2 \geq \sum_{i=1}^{n} t_i.\mathsf{std\_Attr}^2 * x_i^2. \tag{2.2}$$

If tuples have correlation, suppose the correlation of tuple t_i and t_j is stored in table $\mathsf{Covariance}(i, j)$. In this case, instead of Equation (2.2), we add the following quadratic constraint:

$$c^2 \geq \sum_{i=1}^{n} \sum_{j=1}^{n} \mathsf{Covariance}(i, j) * x_i * x_j. \tag{2.3}$$

The final formulation thus includes the pair of Equations (2.1) and (2.2) if all distributions are independent, or else the pair of Equations (2.1) and (2.3).

We now provide an example of how this formulation can be derived. For simplicity, let us consider the following probabilistic constraint:

$$\Pr\left(\sum_{i=1}^{n} T_i x_i \geq v\right) \geq p,$$

where all T_i's are independent Gaussian distributions, with $T_i \sim N[\mu_i, \sigma_i^2]$. First, notice that $\sum_{i=1}^{n} T_i x_i$ is itself a Gaussian random variable, being the sum of Gaussian variables. Let us refer to it as $S_x \sim N[\mu_x, \sigma_x^2]$, with mean $\mu_x = \sum_{i=1}^{n} \mu_i x_i$ and variance $\sigma_x^2 = \sum_{i=1}^{n} \sigma_i^2 x_i^2$. Therefore, the constraint becomes:

$$\Pr\left(S_x \geq v\right) \geq p.$$

Let $F_x(v) = \Pr\left(S_x \leq v\right)$ be the cumulative distribution function of S_x, evaluated at v. Since S_x is Gaussian with mean μ_x and variance σ_x^2, $F_x(v) = \Phi\left(\frac{v-\mu_x}{\sigma_x}\right)$, where Φ is the cumulative distribution function of the *standard* Gaussian distribution. Therefore, the constraint becomes:

$$\begin{aligned}
1 - \Phi\left(\frac{v-\mu_x}{\sigma_x}\right) &\geq p \\
-\Phi\left(\frac{v-\mu_x}{\sigma_x}\right) &\geq -(1-p) \\
-\Phi^{-1}\left(\Phi\left(\frac{v-\mu_x}{\sigma_x}\right)\right) &\geq -\Phi^{-1}(1-p) \\
-\frac{v-\mu_x}{\sigma_x} &\geq \Phi^{-1}(p) \\
\mu_x - v &\geq z_p \sigma_x,
\end{aligned}$$

where we used the facts that Φ is monotonic increasing, $-\Phi^{-1}(1-p) = \Phi^{-1}(p)$, and $\sigma_x > 0$. Rewriting the last derivation in full, we obtain:

$$z_p \sum_{i=1}^{n} \sigma_i^2 x_i^2 \leq \sum_{i=1}^{n} \mu_i x_i - v. \tag{2.4}$$

The two constraints that we add to the linear program according to the previous translation rules are:

$$\begin{cases} z_p c \leq \sum_{i=1}^{n} \mu_i x_i - v \\ c^2 \geq \sum_{i=1}^{n} \sigma_i^2 x_i^2 \end{cases} \tag{2.5}$$

and it is very easy to prove that the two constraints (2.5) together are equivalent to (2.4) alone. In fact, setting $c^2 := \sum_{i=1}^{n} \sigma_i^2 x_i^2$ immediately implies $z_p c \leq \sum_{i=1}^{n} \mu_i x_i - v$ and $c^2 \geq \sum_{i=1}^{n} \sigma_i^2 x_i^2$ from Equation (2.4), and Equation (2.4) is also an immediate consequence of Equations (2.5).

CHAPTER 3

SCALABLE DETERMINISTIC PACKAGE QUERY EVALUATION

In this chapter, we introduce our evaluation methods for package queries on deterministic data. Using the ILP formulation presented in the previous chapter, we first develop **Direct**, our basic evaluation method for package queries, which is more suitable for small datasets. In Section 3.2, we extend this technique to our main algorithm, **SketchRefine**, which supports efficient package evaluation in large datasets.

3.1 Package query evaluation with Direct

Package evaluation with **Direct** employs three steps:

1. **Base relations.** We first compute the base relations, such as R_β, R_c, and R_p, with a series of standard SQL queries, one for each, or by simply scanning R once and populating these relations simultaneously.

2. **ILP formulation.** We transform the PaQL query to an ILP problem using the rules described in Section 2.4.1. After this phase, all variables x_i such that $x_i = 0$ can be eliminated from the ILP problem because the corresponding tuple t_i cannot appear in any package solution. This can significantly reduce the size of the problem.

3. **ILP execution.** We employ an off-the-shelf ILP solver, as a black box, to get a solution to each of the integer variables x_i. Each x_i informs the number of times tuple t_i should be included in the answer package.

Example 6 (ILP Solution)**.** The ILP solver operating on the program of Figure 2.3 returns the variable assignments to x_i that lead to the optimal solution; $x_i = 0$ means that tuple t_i is not included in the output package, and $x_i = k$ means that tuple t_i is included k times in the output package. Thus, the result of $\mathcal{Q}$ is the package: $\{t_2, t_3, t_5\}$.

3.2 Scalable package evaluation with SketchRefine

The **Direct** algorithm has two crucial drawbacks. First, it is only applicable if the input relation is small enough to fit entirely in main memory: ILP solvers, such as IBM's CPLEX, require the entire problem to be loaded in memory before execution. Second, even for problems that fit in main memory, this approach may fail due to the complexity of the integer problem. In fact, integer linear programming is a notoriously hard problem, and modern ILP solvers use algorithms, such as *branch-and-cut* [113], that often perform well in practice, but can "choke" even on small problem sizes due to their exponential worst-case complexity [42]. This may result in unreasonable performance due to solvers using too many resources (main memory, virtual memory, CPU time), eventually thrashing the entire system.

In this section, we present **SketchRefine**, an approximate divide-and-conquer evaluation technique for efficiently answering package queries on large datasets. Rather than solving the original large problem with **Direct**, **SketchRefine** smartly decomposes a query into smaller queries, formulates them as ILP problems, and employs an ILP solver as a black-box evaluation method to answer each individual query. By breaking down the

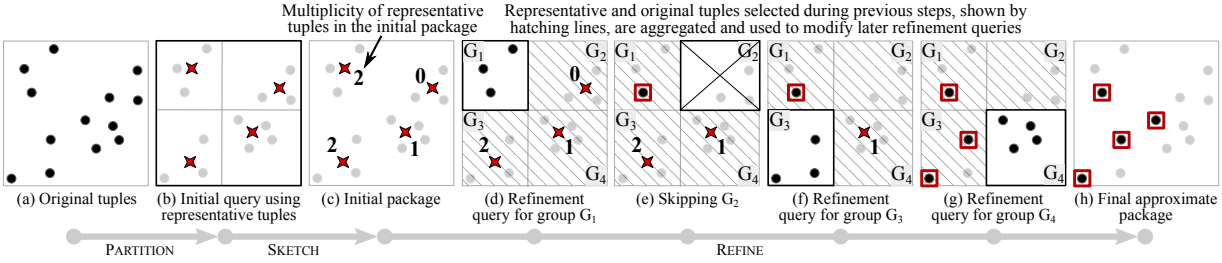


Figure 3.1: The original tuples (a) are partitioned into four groups and a representative is constructed for each group (b). The initial sketch package (c) contains only representative tuples, with possible repetitions up the size of each group. The refine query for group G_1 (d) involves the original tuples from G_1 and the aggregated solutions to all other groups (G_2, G_3, and G_4). Group G_2 can be skipped (e) because no representatives could be picked from it. Any solution to previously refined groups are used while refining the solution for the remaining groups (f and g). The final approximate package (h) contains only original tuples.

problem into smaller subproblems, the algorithm avoids the drawbacks of the **DIRECT** approach. Our implementation of **SKETCHREFINE** uses an ILP solver as its underlining black box for solving the smaller queries; however, **SKETCHREFINE** is more general in that it can be used to scale *any other black-box solution for solving package queries.* Further, we prove that **SKETCHREFINE** is guaranteed to always produce feasible packages with an approximate objective value (Section 3.3.2).

The algorithm is based on an important observation: *similar tuples are likely to be interchangeable within packages*. A group of similar tuples can therefore be "compressed" to a single *representative tuple* for the entire group. **SKETCHREFINE** *sketches* an initial answer package using only the set of representative tuples, which is substantially smaller than the original dataset. This initial solution is then *refined* by evaluating a subproblem for each group, iteratively replacing the representative tuples in the current package solution with original tuples from the dataset. Figure 3.1 provides a high-level illustration of the three main steps of **SKETCHREFINE**:

Algorithm 1 Scalable package query evaluation

```
1: procedure SketchRefine(Q: Package query, P: Partitioning)
2:     p_S ← Sketch(Q, P)
3:     if failure then
4:         return infeasible
5:     else
6:         (p, F) ← Refine(Q, P, p_S)
7:         if F ≠ ∅ then                                  ▷ Refine failure
8:             return infeasible
9:         else                                           ▷ Refine success
10:            return p
```

1. **Offline partitioning (Section 3.2.1).** The algorithm assumes a partitioning of the data into groups of similar tuples. This partitioning is performed offline (not at query time), and our experiments show that **SketchRefine** remains very effective even with partitionings that do not match the query workload (Section 3.4.2.3). In our implementation, we partition data using k-dimensional quad trees [61], but other partitioning schemes are possible.

2. **Sketch (Section 3.2.2.1). SketchRefine** sketches an initial package by evaluating the package query only over the set of representative tuples.

3. **Refine (Section 3.2.2.2).** Finally, **SketchRefine** transforms the initial package into a complete package by replacing each representative tuple with some of the original tuples from the same group, one group at a time.

SketchRefine always constructs *approximate feasible* packages, i.e., packages that satisfy all the query constraints, but with a possibly sub-optimal objective value that is guaranteed to be within certain approximation bounds (Section 3.3.2). **SketchRefine** may suffer from *false infeasibility*, which happens when the algorithm reports a feasible query to be infeasible. The probability of false infeasibility is, however, low and bounded (Section 3.3.3).

In the subsequent discussion, we use $\mathsf{R}(\mathsf{attr}_1, \ldots, \mathsf{attr}_k)$ to denote an input relation with k attributes. R is partitioned into m groups $G_1, \ldots, G_m$. Each group $G_i \subseteq \mathsf{R}$, $1 \leq i \leq m$, has a representative tuple $\tilde{t}_i$, which may not always appear in R. We denote the partitioned space with $\mathcal{P} = \{(G_i, \tilde{t}_i) \mid 1 \leq i \leq m\}$. We refer to packages that contain representative tuples as *sketch packages* and packages with only original tuples as *complete packages* (or simply *packages*). We denote a complete package with p and a sketch package with $p_{\mathcal{S}}$, where $\mathcal{S} \subseteq \mathcal{P}$ is the set of groups that are yet to be refined to transform $p_{\mathcal{S}}$ to a complete answer package p.

3.2.1 Offline partitioning

SketchRefine relies on an offline partitioning of the input relation R into groups of similar tuples. Partitioning is based on a set of *partitioning attributes* from the input relation R, a *size* threshold, and a set of *diameter* bounds. The partitioning attributes can be any subset of the numerical attributes of R.

Definition 1 (Size threshold, τ)**.** *The* size threshold τ, $1 \leq \tau \leq n$, *restricts the size of each partitioning group* G_i, $1 \leq i \leq m$, *to a maximum of* τ *original tuples, i.e.,* $|G_i| \leq \tau$.

Definition 2 (Diameter bounds)**.** *The* diameter $d_{ij} \geq 0$ *of a group* G_i, $1 \leq i \leq m$, *on attribute* attr_j, $1 \leq j \leq k$, *is the greatest absolute distance between all pairs of tuples within group* G_i*:*

$$d_{ij} = \max_{t_1, t_2 \in G_i} |t_1.\mathsf{attr}_j - t_2.\mathsf{attr}_j| \tag{3.1}$$

The diameter bounds $\omega_{ij} \geq 0$, $1 \leq i \leq m$, $1 \leq j \leq k$, *require all diameters to be bounded by* $d_{ij} \leq \omega_{ij}$.

The size threshold, τ, affects the number of partitions, m: a lower τ leads to smaller partitions, but more of them (larger m). As we discuss later (Section 3.2.2), for best response time of **SketchRefine**, τ should be set so that both m and τ are small. Our

experiments show that a proper setting can lead to an order of magnitude improvement in query response time (Section 3.4.2.2).

The diameter bounds, ω_{ij}, are not required, but they can be enforced to ensure a desired approximation guarantee (Section 3.3.2). Note that the same partitioning can be used to support a multitude of queries over the same dataset. In our experiments, we show that a single partitioning performs consistently well across different queries. In general, enforcing the diameter limits may cause the resulting partitions to become excessively small. While still obeying the approximation guarantees, this could increase the number of resulting partitions and thus degrade the running time performance of **SketchRefine**. This is an important trade-off between running time and quality that we also observe in our experiments (Section 3.4.2.4), and it is a very common characteristic of most approximation schemes [159].

Partitioning method. Different methods can be used for partitioning. Our implementation is based on k-dimensional *quad-tree indexing* [61]. The method recursively partitions a relation into groups until all the groups satisfy the size threshold and meet the diameter limits. First, relation R is augmented with an extra group ID column gid, such that $t.\mathsf{gid} = i$ iff tuple t is assigned to group G_i. The procedure initially creates a single group G_1 that includes all the original tuples from relation R, by initializing $\mathsf{gid} = 1$ for all tuples. Then, it recursively proceeds as follows:

- The procedure computes the sizes and diameters of the current groups via a query that groups tuples by their gid value. The same group-by query also computes the *centroid* tuple of each group. The centroid is computed by averaging the tuples in the group on each of the partitioning attributes.

- If group G_i has more tuples than the size threshold, or a diameter larger than the allowed bound, the tuples in group G_i are partitioned into 2^k subgroups, where k is the number of partitioning attributes. The group's centroid is the split point to generate sub-partitions: tuples that reside in the same sub-partition are grouped together.

Our method recursively executes two SQL queries on each subgroup that violates the size or the diameter conditions.

Stored representatives. After partitioning, a group-by query computes the minimum, maximum, and average values of all the partitioning attributes, and stores them in a relational table. At query time, the algorithm loads representatives from this table, selecting only one aggregate type per query attribute (either minimum, maximum of average), into a *representative relation* $\tilde{R}(\mathsf{gid}, \mathsf{attr}_1, \ldots, \mathsf{attr}_k)$. To ensure approximation guarantees (Section 3.3.2), the maximum (minimum, resp.) value is chosen for a maximization (minimization, resp.) query. For all other attributes, the algorithm picks the average value.

Alternative partitioning approaches. We experimented with different clustering algorithms, such as *k-means* [72], *hierarchical clustering* [89] and DBSCAN [56], using off-the-shelf libraries such as *Scikit-learn* [116]. Existing clustering algorithms present various problems: First, they tend to vary substantially in the properties of the generated clusters. In particular, none of the existing clustering techniques can natively generate clusters that satisfy the size threshold τ and diameter limits ω_{ij}. In fact, most of the clustering algorithms take as input the *number of clusters* to generate, without offering any means to restrict the size of each cluster nor their diameter. Second, existing implementations only support in-memory cluster computation, and DBMS-oriented implementations usually need complex and inefficient queries. On the other hand, space partitioning techniques

from multi-dimensional indexing, such as *k-d trees* [15] and *quad trees* [61], can be more easily adapted to satisfy the size and diameter limits, and to work within the database: our partitioning method works directly on the input table via simple SQL queries.

Finally, partitioning could be dynamically generated at query time: By maintaining the entire hierarchical structure of the quad-tree index, one can traverse the index at query time to generate the coarsest partitioning that satisfies the required size and diameter limits. However, index traversal incurs additional overhead at query time, compared to using a precomputed static partitioning.

One-time cost. Partitioning is an expensive procedure. To avoid paying its cost at query time, the dataset is partitioned in advance and used to answer a workload of package queries. For a known workload, our experiments show that partitioning the dataset on the union of all query attributes provides the best performance in terms of query evaluation time and approximation error for the computed answer package (Section 3.4.2.3). We also demonstrate that our query evaluation approach is robust to a wide range of partition sizes, and to imperfect partitions that cover more or fewer attributes than those used in a particular query. This means that, even without a known workload, a partitioning performed on all of the data attributes still provides good performance.

Enforcing a diameter limit guarantees the theoretical approximation bounds of **SketchRefine** (Section 3.3.2). However, partitioning only with a size threshold can also achieve good quality in practice: Since partitioning splits a group on its centroid, the resulting sub-partitions will naturally have smaller diameters. Our experiments (Section 5.2.2) show that partitioning on a size threshold alone results in good approximations while reducing the offline partitioning cost: Meeting a size threshold requires fewer

partitioning iterations than meeting a diameter limit especially if the dataset is sparse across the attribute domains.

3.2.2 Query evaluation with SketchRefine

During query evaluation, **SketchRefine** first *sketches* a package solution using the representative tuples (**Sketch**), and then it *refines* it by replacing representative tuples with original tuples (**Refine**). We describe these steps using the example query $\mathcal{Q}$ from Figure 2.1.

3.2.2.1 Sketch

Using the representative relation $\tilde{R}$ (Section 3.2.1), the **Sketch** procedure constructs and evaluates a *sketch query*, $\mathcal{Q}(\tilde{\mathsf{R}})$. The result is an initial sketch package, $p_{\mathcal{S}}$, containing representative tuples that satisfy the same constraints as the original query $\mathcal{Q}$:

$\mathcal{Q}(\tilde{\mathsf{R}})$: SELECT PACKAGE$(*)$ AS $p_{\mathcal{S}}$
FROM $\tilde{\mathsf{R}}$
WHERE $\tilde{\mathsf{R}}$.gluten $= 0$
SUCH THAT
COUNT$(p_{\mathcal{S}}.*) = 3$ AND
SUM$(p_{\mathcal{S}}$.kcal) BETWEEN 2.0 AND 2.5 AND
(SELECT COUNT$(*)$ **FROM** $p_{\mathcal{S}}$ **WHERE gid** $= 1) \leq |G_1|$
AND ...
(SELECT COUNT$(*)$ **FROM** $p_{\mathcal{S}}$ **WHERE gid** $= m) \leq |G_m|$
MINIMIZE SUM$(p_{\mathcal{S}}$.sat_fat)

The new global constraints, highlighted in bold, ensure that every representative tuple does not appear in $p_{\mathcal{S}}$ more times than the size of its group, G_i. This accounts for the repetition constraint REPEAT 0 in the original query. Generalizing, with REPEAT ρ, each $\tilde{t}_i$ can be repeated up to $|G_i|(1+\rho)$ times. These constraints are simply omitted from $\mathcal{Q}(\tilde{\mathsf{R}})$ if the original query does not contain a repetition constraint.

Since the representative relation $\tilde{R}$ contains exactly m representative tuples, the ILP problem corresponding to this query has only m variables. This is typically small enough for the black-box ILP solver to manage directly, and thus we can solve this package query using the **Direct** method (Section 3.1). If m is too large, we can solve this query *recursively* with **SketchRefine**: the set of m representatives is further partitioned into smaller groups until the subproblems reach a size that can be efficiently solved directly.

The **Sketch** procedure *fails* if the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ is infeasible, in which case **SketchRefine** reports the original query $\mathcal{Q}$ as infeasible (Algorithm 1). This may constitute *false infeasibility*, if $\mathcal{Q}$ is actually feasible. In Section 3.3.3, we show that the probability of false infeasibility is low and bounded, and we present simple methods to avoid this outcome.

3.2.2.2 Refine

Using the sketched solution over the representative tuples, the **Refine** procedure iteratively replaces the representative tuples with tuples from the original relation R, until no more representatives are present in the package. The algorithm *refines* the sketch package $p_{\mathcal{S}}$ one group at a time. For a group G_i with representative $\tilde{t}_i$, let $\tilde{p}_i \subseteq p_{\mathcal{S}}$ be the set of representatives picked from G_i (i.e., $\tilde{t}_i$ with possible duplicates). The algorithm proceeds as follows:

- It derives package $\bar{p}_i$ from $p_{\mathcal{S}}$, by eliminating all instances of $\tilde{t}_i$ from $p_{\mathcal{S}}$. That is, $\bar{p}_i = p_{\mathcal{S}} \setminus \tilde{p}_i$. This is a solution to all groups except G_i.
- The algorithm then constructs a *refine query*, $\mathcal{Q}_i(p_{\mathcal{S}})$, which searches for a set of tuples $p_i \subseteq G_i$ to replace the eliminated representatives:

$\mathcal{Q}_i(p_{\mathcal{S}})$: SELECT PACKAGE$(*)$ AS p_i
FROM G_i REPEAT 0
WHERE G_i.gluten $= 0$

SUCH THAT
COUNT(p_i.*) + **COUNT**($\bar{p}_i$.*) = 3 AND
SUM(p_i.kcal) + **SUM**($\bar{p}_i$.kcal) BETWEEN 2.0 AND 2.5
MINIMIZE SUM(p_i.sat_fat)

- The algorithm adds the result of $\mathcal{Q}_i(p_{\mathcal{S}})$, p_i, in the current solution, $p_{\mathcal{S}}$. Now, group G_i is *refined* with actual tuples.

In $\mathcal{Q}_i(p_{\mathcal{S}})$, COUNT($\bar{p}_i$.*) and SUM($\bar{p}_i$.kcal) are values computed directly on $\bar{p}_i$ before the query is formed. They are used to modify the original constraint bounds to account for tuples and representatives already chosen for all the other groups. The global constraints in $\mathcal{Q}_i(p_{\mathcal{S}})$ ensure that the combination of tuples in p_i and $\bar{p}_i$ satisfy the original query $\mathcal{Q}$. Thus, this step produces the new *refined sketch package* $p'_{\mathcal{S}'} = \bar{p}_i \cup p_i$, where $\mathcal{S}' = \mathcal{S} \setminus \{(G_i, \tilde{t}_i)\}$.

Since G_i has at most τ tuples, the ILP problem corresponding to $\mathcal{Q}_i(p_{\mathcal{S}})$ has at most τ variables. This is typically small enough for the black-box ILP solver to solve directly, and thus we can solve this package query using the **Direct** method (Section 3.1). Similarly to the sketch query, if τ is too large, we can solve this query recursively with **SketchRefine**: the tuples in group G_i are further partitioned into smaller groups until the subproblems reach a size that can be efficiently solved directly.

Ideally, the **Refine** step will only process each group with representatives in the initial sketch package once. However, the order of refinement matters as each refinement step is greedy: it selects tuples to replace the representatives of a single group, without considering the effects of this choice on other groups. As a result, a particular refinement step may render the query infeasible (no tuples from the remaining groups can satisfy the constraints). When this occurs, **Refine** employs a *greedy backtracking* strategy that reconsiders groups in a different order.

Greedy-backtracking REFINE. **REFINE** activates backtracking when it encounters an infeasible *refine query*, $\mathcal{Q}_i(p_{\mathcal{S}})$. Backtracking *greedily prioritizes* the infeasible groups. This choice is motivated by a simple heuristic: if the refinement on G_i fails, it is likely due to choices made by previous refinements; therefore, by prioritizing G_i, we reduce the impact of other groups on the feasibility of $\mathcal{Q}_i(p_{\mathcal{S}})$. This heuristic does not affect the approximation guarantees (Section 3.3.2).

Algorithm 2 details the **REFINE** procedure. The algorithm logically traverses a *search tree* (which is never constructed, but is the result of recursive calls and bactracking), where each node corresponds to a unique sketch package $p_{\mathcal{S}}$. The traversal starts from the *root*, corresponding to the initial sketch package, where no groups have been refined ($\mathcal{S} = \mathcal{P}$), and finishes at the first encountered *leaf*, corresponding to a complete package ($\mathcal{S} = \emptyset$). The algorithm terminates as soon as it encounters a complete package, which it returns (line 4). The algorithm maintains a set of failed groups, $\mathcal{F}$, initially empty (line 2), and assumes a (initially random) refinement order for all groups in $\mathcal{S}$, stored in a priority queue $\mathcal{U}$ (line 6). It then tries to solve the refine query corresponding to each of the groups in the queue (line 12). When a refine query succeeds, the algorithm recursively proceeds with the next group in the queue (lines 13-18). If any of the refine queries fails, the failing group is added to $\mathcal{F}$, and the algorithm immediately backtracks, reporting the failure to the parent node in the search tree (lines 25-29). Failures can occur at any depth of the traversal. If a recursive call fails, all the failing groups ($\mathcal{F}'$) are prioritized (lines 19-22). The package produced by **REFINE**, if any, is always guaranteed to be feasible (see Section 3.3.1).

Let $T(\tau)$ be the time taken by the black box (in our case, **DIRECT** using an ILP solver) to solve a problem of size τ. We express the time complexity of the refine procedure as a function of $T(\tau)$ and m, the number of partitions used by **SKETCHREFINE**. In the best case, all refine queries are feasible and the algorithm never backtracks. In this case, the

algorithm makes up to m calls to the solver to solve problems of size up to τ, one for each refining group. In the worst case, **SketchRefine** tries every group ordering leading to a factorial number of calls to the solver, $O(T(\tau)m!)$. Our experiments show that the best case is the most common and backtracking occurs infrequently.

False infeasibility and hybrid sketch queries. For a feasible query $\mathcal{Q}$, false negatives, or *false infeasibility*, may happen in two cases: (1) when the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ is infeasible; (2) when greedy backtracking fails (possibly due to suboptimal partitioning). In both cases, **SketchRefine** would (incorrectly) report a feasible package query as infeasible. False negatives are, however, extremely rare, as Theorem 3 establishes in Section 3.3.3.

In our evaluation, we use a small heuristic modification to **SketchRefine** to deal with these cases, which creates a hybrid query by merging the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ with one of the refine queries. The *hybrid* sketch query, executed in place of the original sketch query, selects tuples from a group and, at the same time, representative tuples from all the remaining groups. This simple technique can greatly reduce false infeasibility by circumventing three potential cases of failure: (1) The original sketch query, $\mathcal{Q}(\tilde{\mathsf{R}})$, may be infeasible due to a bad representative from one of the groups. An hybrid sketch query over that group could render the sketch phase possible. (2) If a group fails in a later refine stage, solving that group upfront with a hybrid sketch query could render the group's problem feasible, thanks to having representatives for the other groups. (3) If a group fails in a later refine stage, a hybrid sketch query on a different group could avoid selecting representatives for the failing group altogether. The algorithm tries a hybrid sketch query on each group whenever the original sketch query is infeasible or when all refines fail; it then proceeds normally if one of the hybrid queries is feasible. Hybrid sketch proves extremely effective on our experimental workload (Section 5.2.2): **SketchRefine** with

hybrid sketch does not encounter even a single case of false infeasibility, i.e., there is no query for which **Direct** produces a solution but **SketchRefine** does not.

3.3 Theoretical analysis of SketchRefine

SketchRefine scales package evaluation by breaking the problem into smaller, manageable subproblems: the **Sketch** phase evaluates a package query over the representative tuples of the partitions, and the **Refine** phase evaluates package queries over each partition. This scalability comes at the price of accuracy. A package returned by **SketchRefine** is guaranteed to satisfy all the query constraints (Section 3.3.1), but it may have a worse objective value than the package produced by **Direct** evaluation. Moreover, **SketchRefine** may incorrectly determine that a package query is infeasible, when in fact it has a solution (false infeasibility). In this section, we provide a theoretical analysis of the quality of results produced by **SketchRefine**. Specifically, we present two theoretical results. First, we show that **SketchRefine** offers strong approximation guarantees: a package produced by **SketchRefine** is guaranteed to be within a $(1 \pm \epsilon)$-factor from the package produced by **Direct**. Second, we show that **SketchRefine** fails to produce a package to a feasible query (false infeasibility) with low probability.

3.3.1 Correctness of SketchRefine

Proposition 1 (Correctness of **Refine**)**.** *A package produced by* **Refine** *is guaranteed to satisfy the query constraints.*

Proof. The proposition follows from the fact that, by construction, the refine query, $\mathcal{Q}_i(p_S)$, identifies tuples replacements for the representatives that do not break the overall constraints of the original query. ■

As a direct consequence of the proposition, if **SketchRefine** returns a package, the package is guaranteed to satisfy all the query constraint since **Refine** is the final step of the algorithm.

3.3.2 Approximation guarantees

Direct and **SketchRefine** employ a black-box solver to evaluate either the original query (**Direct**), or the subqueries (the sketch and refine queries of **SketchRefine**). If the solver is exact, then **Direct** returns optimal solutions, and the approximation guarantees of **SketchRefine** are with respect to the true optimal. In general however, solvers may not be exact (e.g., ILP solvers typically provide approximations), in which case the approximation bound of **SketchRefine** is with respect to the approximation of the solver. **SketchRefine** allows control of its approximation bounds through its offline partitioning. Specifically, we prove that, for a desired approximation parameter ϵ, we can derive diameter bounds ω_{ij} (for each partitioning group G_i and attribute attr_j) for the offline partitioning that guarantee that the solution produced by **SketchRefine** (if any) has objective value $(1 \pm \epsilon)$-factor close to the objective value of the solution produced by the solver for the same query.

Theorem 2 (Approximation Bounds)**.** *Let* $\mathsf{R}(\mathsf{attr}_1, \ldots, \mathsf{attr}_k)$ *be a relation with* k *attributes, and let* $\mathcal{Q}$ *be a feasible package query with a maximization (minimization, resp.) objective over* R*. Let* S *be an exact solver that produces an answer to* $\mathcal{Q}$ *with optimal objective value* OPT*. We denote with* ALG *the objective value of the package returned by* **SketchRefine** *using* S *as a black-box solver. For any* $\epsilon \in [0,1)$ *(*$\epsilon \in [0,\infty)$*, resp.), there exists* $\beta \in [0,1)$ *(*$\beta \in [1,\infty)$*, resp.) that depends on* ϵ*, such that if* R *is partitioned into* m *groups with diameter limits:*

$$\omega_{ij} = \min_{t \in G_i} \{|1-\beta| \cdot |t.\mathsf{attr}_j|\}, \quad \forall i \in [1,m],\, \forall j \in [1,k] \tag{3.2}$$

then $ALG \geq (1-\epsilon)OPT$ *(*$ALG \leq (1+\epsilon)OPT$*, resp.).*

We present the proof of the theorem for the case of maximization queries. The minimization case follows analogous reasoning. Without loss of generality, we consider a feasible package query $\mathcal{Q}$ with a summation constraint on each of the k attributes, $\mathsf{SUM}(\mathsf{attr}_j) \leq U_j$, $j \in [1,k]$, and a maximization objective on $\mathsf{SUM}(\mathsf{attr}_{obj})$. A COUNT constraint is a special case of a SUM over an attribute that is equal to 1. Partitioning over this attribute would result in groups with zero diameter (the value of the attribute for all tuples in the group is the same). Therefore, with respect to this attribute, representatives are exact. Essentially, COUNT constraints do not affect the approximation of the result.

We prove Theorem 2 in two steps. First, we show that the initial **Sketch** package approximates the optimal package by a factor β. Second, we show that the final package returned by the **Refine** procedure approximates the initial **Sketch** package by a factor β as well. Thus, the final result of **SketchRefine** approximates the optimal package by a factor of β^2. We conclude the proof by showing an explicit value for β as a function of ϵ. The proof requires two lemmas (Lemma 2 and Lemma 3 below). The first lemma shows that if a package satisfies $\mathcal{Q}$, replacing the tuples in the package with their representative tuples generates a package that satisfies a relaxed version of $\mathcal{Q}$, where each constraint is relaxed by a factor β. Below, we define such relaxed queries as *β-relaxations*. The second lemma shows that if a package p_1 optimizes $\mathcal{Q}$ and another package p_2 optimizes its β-relaxation, then the objective value of p_1 cannot be worse than the objective value of p_2 by more than a factor β.

We first introduce some needed notation and definitions. Given a package p, we denote the summation of its tuples on attribute attr with $\mathsf{SUM}(p.\mathsf{attr})$, and its objective value

with $\mathsf{OBJ}(p)$, where $\mathsf{OBJ}(p) = \mathsf{SUM}(p.\mathsf{attr}_{obj})$. We now proceed to define the concepts of *ordering*, *feasible*, *optimal*, and *approximate* packages, that are at the core of the proof.

Definition 3 (Package ordering $\succeq$)**.** *A package* p_1 dominates *a package* p_2*, denoted by* $p_1 \succeq p_2$*, iff the objective value of* p_1 *is at least as good as the objective value of* p_2*:* $\mathit{OBJ}(p_1) \geq \mathit{OBJ}(p_2)$*. With slight abuse of notation, we write* $p_1 \succeq \beta\, p_2$ *to denote that the objective value of* p_1 *is as least as good as the objective value of* p_2 *by a factor* β*.*

Definition 4 (Feasible package $\models$)**.** *We say that a package* p *is* feasible *for* $\mathcal{Q}$*, denoted by* $p \models \mathcal{Q}$*, iff for all* $1 \leq j \leq k$*:* $\mathit{SUM}(p.\mathit{attr}_j) \leq U_j$*.*

Definition 5 (Optimal package $\models^*$)**.** *A package* p *is* optimal *for* $\mathcal{Q}$*, denoted by* $p \models^* \mathcal{Q}$*, iff* $p \models \mathcal{Q}$ *and for all* $p' \models \mathcal{Q}$*,* $p \succeq p'$*.*

Definition 6 (β-approximation)**.** *A package* p *is a* β-approximation *for query* $\mathcal{Q}$ *if* $p \models \mathcal{Q}$ *and for all* $p' \models \mathcal{Q}$*,* $p \succeq \beta\, p'$*.*

Definition 7 (β-relaxation)**.** *The* β-relaxation *of query* $\mathcal{Q}$*, denoted by* $\mathcal{Q}_\beta$*, is a query with the same objective function as* $\mathcal{Q}$*, and with* k *global constraints, for all* $1 \leq j \leq k$*:*

$$\mathit{SUM}(\mathsf{attr}_j) \leq \beta^{-1} U_j$$

Definition 8 (Representative projection π)**.** *The representative projection of a package* p*, denoted by* $\pi(p)$*, is a function that substitutes each tuple in* p *with its representative tuple.*

Because representative tuples have the best value on the objective attribute attr_{obj} of all the tuples in its group, π satisfies the following property:

Property 1. *The representative projection of a package dominates the package:* $\pi(p) \succeq p$*.*

Before stating Lemma 2, we introduce another intermediate result. Lemma 1 states that the diameter conditions of Equation (3.2) guarantee that all the tuples in a group are "close" to each other by a factor no larger than β. We refer to this as β-*closeness*, and we generalize this concept to pairs of packages: two packages are β-close to each other if their sums (on any attribute) are close to each other by a factor β.

Definition 9 (β-CLOSENESS)**.** *Any two tuples t_1 and t_2 are β-close to each other iff for all $1 \leq j \leq k$:*

$$t_1.\mathsf{attr}_j \geq \beta\, t_2.\mathsf{attr}_j \;\; \textit{and} \;\; t_2.\mathsf{attr}_j \geq \beta\, t_1.\mathsf{attr}_j$$

Any two packages p_1 and p_2 are β-close to each other iff for all $1 \leq j \leq k$:

$$\mathit{SUM}(p_1.\mathit{attr}_j) \geq \beta\, \mathit{SUM}(p_2.\mathit{attr}_j)$$

$$\textit{and}$$

$$\mathit{SUM}(p_2.\mathit{attr}_j) \geq \beta\, \mathit{SUM}(p_1.\mathit{attr}_j)$$

Lemma 1. *If the partitioning satisfies the diameter limits of Equation* (3.2)*, then all tuples within the same group are β-close to each other.*

Proof. Consider any group G_i, any attribute attr_j, any pair of tuples t_1, t_2 in G_i. First, $|1-\beta| = (1-\beta)$ as $\beta \in [0,1)$. By Equation (3.1), $t_1.\mathsf{attr}_j \geq t_2.\mathsf{attr}_j - d_{ij}$. By Equation (3.2), $d_{ij} \leq (1-\beta)|t_2.\mathsf{attr}_j|$. Thus, either (i) $-d_{ij} \geq (1-\beta)\, t_2.\mathsf{attr}_j$ or (ii) $-d_{ij} \geq (\beta-1)\, t_2.\mathsf{attr}_j$:

$$\text{If (i):} \;\; t_1.\mathsf{attr}_j \geq t_2.\mathsf{attr}_j + (1-\beta)\, t_2.\mathsf{attr}_j > \beta\, t_2.\mathsf{attr}_j,$$

$$\text{If (ii):} \;\; t_1.\mathsf{attr}_j \geq t_2.\mathsf{attr}_j + (\beta-1)\, t_2.\mathsf{attr}_j = \beta\, t_2.\mathsf{attr}_j,$$

which implies the lemma. ■

The following lemma states that the representative projection of a feasible package for query $\mathcal{Q}$ satisfies a β-relaxed version of the same query.

Lemma 2 (Representative projection relaxation)**.** *For any package p: $p \models \mathcal{Q} \implies \pi(p) \models \mathcal{Q}_\beta$.*

Proof. By hypothesis, for all $1 \leq j \leq k$, $U_j \geq \mathsf{SUM}(()p.\mathsf{attr}_j)$. By Lemma 1, $\mathsf{SUM}(()p.\mathsf{attr}_j) \geq \beta\, \mathsf{SUM}(()\pi(p).\mathsf{attr}_j)$. Therefore, $\mathsf{SUM}(()\pi(p).\mathsf{attr}_j) \leq \beta^{-1}\, U_j$. ■

Lemma 3 (β-relaxation approximation)**.** *For any packages p_1, p_2: $p_1 \models^* \mathcal{Q}$ and $p_2 \models^* \mathcal{Q}_\beta \implies p_1 \succeq \beta\, p_2$.*

Proof. Because $p_2 \models \mathcal{Q}_\beta$, for all $1 \leq j \leq m$, $\mathsf{SUM}(p_2.\mathsf{attr}_j) \leq \beta^{-1} U_j$. Thus, $\beta\, \mathsf{SUM}(p_2.\mathsf{attr}_j) \leq U_j$ and therefore, with abuse of notation, $\beta\, p_2 \models \mathcal{Q}$. Since $p_1 \models^* \mathcal{Q}$, $p_1 \succeq \beta\, p_2$. ■

We are now ready to prove Theorem 2.

Proof of Theorem 2. Let the initial sketch package be denoted by $p^{(0)}$. Suppose, without loss of generality, that the algorithm refines the initial package in the order: $G_1, G_2, \ldots, G_m$. Let $p^{(i)}$ denote the intermediate refined package produced at the i-th iteration of the algorithm. The final complete package returned by the algorithm is thus $p^{(m)}$. Let $p^* \models^* \mathcal{Q}$ be an optimal package. To prove the theorem, we show that there exists a β such that $p^{(m)} \succeq \beta^2\, p^*$. We do so in two steps:

$$p^{(0)} \succeq \beta\ p^* \quad (\textbf{Sketch}) \qquad\qquad p^{(m)} \succeq \beta\ p^{(0)} \quad (\textbf{Refine})$$

(**Sketch**) First, notice that $p^{(0)} \models^* \mathcal{Q}$ because $p^{(0)}$ optimizes the **Sketch** query $\mathcal{Q}(\tilde{\mathsf{R}})$ (Section 3.2.2.1), which has identical constraints and maximization objective as $\mathcal{Q}$. Consider $p' \models^* \mathcal{Q}_\beta$, the optimal package for the relaxed query $\mathcal{Q}_\beta$ constructed with representative

tuples. By Lemma 3, we know that $p^{(0)} \succeq \beta\, p'$. By Lemma 2, we also know that $\pi(p^*) \models \mathcal{Q}_\beta$. Since p' is the optimal package for $\mathcal{Q}_\beta$, $p' \succeq \pi(p^*)$. Finally, by Property 1 of π, we also know that $\pi(p^*) \succeq p^*$. Putting these together, we have that:

$$p^{(0)} \succeq \beta\, p' \succeq \beta\, \pi(p^*) \succeq \beta\, p^*$$

(**Refine**) Consider package $p_i^{(i)}$, the solution the i-th **Refine** query (Section 3.2.2.2) computed at the i-th iteration of the algorithm. Clearly, $p_i^{(i)} \models^* \mathcal{Q}_i(p_{\mathcal{S}})$ because it optimizes the **Refine** query. **SketchRefine** maintains this solution for group G_i until the end of the procedure, thus $p_i^{(m)} = p_i^{(i)}$ and, therefore, $p_i^{(m)} \models^* \mathcal{Q}_i(p_{\mathcal{S}})$. Consider now $p_i^{(0)}$, the set of representatives computed during the **Sketch** phase for group G_i. Because of Lemma 1, during the course of the algorithm, the constraints of a **Refine** query can only vary by a factor β. Thus, it must be that $p_i^{(0)} \models \mathcal{Q}_i(p_{\mathcal{S}})_\beta$. Let $p'' \models^* \mathcal{Q}_i(p_{\mathcal{S}})_\beta$ be the optimal package for the relaxed version of $\mathcal{Q}_i(p_{\mathcal{S}})$. Then, $p'' \succeq p_i^{(0)}$. Also, by Lemma 3, we know that $p_i^{(m)} \succeq \beta\, p''$. Putting these together, we have that:

$$p_i^{(m)} \succeq \beta\, p'' \succeq \beta\, p_i^{(0)}.$$

Finally, because $p^{(m)} = \sum_{i=1}^{m} p_i^{(m)}$ and $p^{(0)} = \sum_{i=1}^{m} p_i^{(0)}$, by linearity of sum we have that $p^{(m)} \succeq \beta\, p^{(0)}$.

Thus, for $\beta = (1-\epsilon)^{\frac{1}{2}}$ ($\beta = (1+\epsilon)^{\frac{1}{2}}$, resp.), we get approximation factor $1-\epsilon$ ($1+\epsilon$, resp.). ■

The theorem implies that, in order to obtain $(1 \pm \epsilon)$-factor approximation, the partitioning must satisfy the following diameter conditions for each group G_i and attribute attr_j:

$$\omega_{ij} = \begin{cases} \min\limits_{t \in G_i} |1-(1-\epsilon)^{\frac{1}{2}}| \cdot |t.\mathsf{attr}_j| & \text{for maximization} \\ \min\limits_{t \in G_i} |1-(1+\epsilon)^{\frac{1}{2}}| \cdot |t.\mathsf{attr}_j| & \text{for minimization} \end{cases}$$

3.3.3 False infeasibility bounds

The following theorem establishes that the probability that **SketchRefine** will fail to find a solution to a feasible query is low and bounded.

Theorem 3. *For any query $\mathcal{Q}$ and any random package P, if $\mathsf{P} \models \mathcal{Q}$, then with high probability: (1) the* **Sketch** *query $\mathcal{Q}(\tilde{\mathsf{R}})$ is feasible; (2) all* **Refine** *queries $\mathcal{Q}_i(p_S)$, $1 \leq i \leq m$, are feasible. Thus,* **SketchRefine** *returns a feasible result.*

Proof. (1) We first show that the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ is feasible with high probability.

Suppose, by hypothesis, that $\mathsf{P} \models \mathcal{Q}$. Thus, P satisfies all constraints of $\mathcal{Q}$. Let $\mathsf{SUM}(\mathsf{A})$ be *any* such constraint, where A is either a constant, an attribute from the schema of the input relation R, or a linear combination of attributes of R. Because P is random, its representative projection $\pi(\mathsf{P})$ (Definition 8), constructed from P by replacing tuples with representatives, is also a random package. Thus, both $\mathsf{SUM}(\mathsf{P.A})$ and $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$ are random variables. We show that, with high probability, $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$ does not differ from the expected $\mathsf{SUM}(\mathsf{P.A})$ and, thus, since P is feasible, so is $\pi(\mathsf{P})$. This implies that the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ is feasible with high probability, as at least one solution to it exists, namely $\pi(\mathsf{P})$.

As a first step, we apply Hoeffding's inequality [74] to $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$. For all $c > 0$, let $\gamma_{c,\mathsf{P}} = 2\exp\left(-\frac{2c^2}{|\mathsf{P}|(\mathsf{MAX(A)}-\mathsf{MIN(A)})^2}\right)$. Hoeffding's inequality establishes that the probability

of $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$ deviating from its expectation by more than c is bounded by a term, $\gamma_{c,\mathsf{P}}$, that is exponentially small in c and $|\mathsf{P}|$:

$$Pr\left[|\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A}) - E[\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})]| \geq c\right] \leq \gamma_{c,\mathsf{P}} \tag{3.3}$$

Let A be the random variable corresponding to the value of attribute A of a random tuple in P, and let $E[A]$ be its expected value. Similarly, let $\tilde{A}$ be the random variable corresponding to a random representative tuple in $\pi(\mathsf{P})$, and $E[\tilde{A}]$ its expected value. Finally, let G be the group a random representative tuple in $\pi(\mathsf{P})$ belongs to. Because representative tuples are the centroids (mean) of all the tuples in their group along the attributes involved in the constraints, we have that:

$$E[\tilde{A}] = E\left[\tfrac{1}{|G|}\textstyle\sum_G A\right] = \tfrac{1}{|G|}\textstyle\sum_G E[A] = E[A] \tag{3.4}$$

The expected sum over package $\pi(\mathsf{P})$ is therefore:

$$E[\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})] = \textstyle\sum_{\mathsf{P}} E[\tilde{A}] = \sum_{\mathsf{P}} E[A] = E[\mathsf{SUM}(\mathsf{P}.\mathsf{A})]$$

Thus Equation (3.3) becomes:

$$Pr\left[|\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A}) - E[\mathsf{SUM}(\mathsf{P}.\mathsf{A})]| \geq c\right] \leq \gamma_{c,\mathsf{P}} \tag{3.5}$$

Equation (3.5) shows that the probability that the sum of A over $\pi(\mathsf{P})$ differs from the expected sum over P by more than $c > 0$ is bounded. Since $\mathsf{SUM}(\mathsf{P}.\mathsf{A})$ is feasible (by hypothesis), so is $\mathsf{SUM}(\pi(\mathsf{P}.)\mathsf{A})$, and the sketch query is feasible on this constraint with

high probability. This is independently true for all query constraints. Thus, the probability of the overall sketch query being infeasible is one minus the probability of all constraints being feasible. With k constraints, this probability (sketch being infeasible) is small and bounded by $1-(1-\gamma_{c,\mathsf{P}})^k$. This term is exponentially small in c and $|\mathsf{P}|$, so, with high probability, the sketch query $\mathcal{Q}_i(p_{\mathcal{S}})$ is feasible.

(2) Now, we show that all refine queries are feasible with high probability. Equation 3.4 allows reasoning about each refine query independently, as replacing representatives with tuples does not change the expected sum in each group.

Let P_i be the tuples in P that belong to group G_i. Then, $\pi(\mathsf{P}_i)$ is the set of representatives in $\pi(\mathsf{P})$ that belong to group G_i. We apply Hoeffding's inequality on $\mathsf{SUM}(\mathsf{P}_i.\mathsf{A})$, obtaining an equation similar to Equation (3.3). The proof now follows the same steps as the proof of (1), now applied on $\mathsf{SUM}(\mathsf{P}_i.\mathsf{A})$. From Equation (3.4), we have that $E[\mathsf{SUM}(\mathsf{P}_i).\mathsf{A})] = E[\mathsf{SUM}(\pi(\mathsf{P}_i))]$. This results in an equation similar to (3.5), showing that, if $\pi(\mathsf{P}_i)$ is feasible for the i-th refine query, then P_i must also be feasible for the same query. When $\pi(\mathsf{P})$ is feasible, $\pi(\mathsf{P}_i)$ is a feasible package for the i-th refine query $\mathcal{Q}_i(p_{\mathcal{S}})$, otherwise the sketch query would be infeasible. This is independently true for all constraints, and the probability of the overall query being infeasible, with k constraints, is bounded by $1-(1-\gamma_{c,\mathsf{P}})^k$. Thus, for every group G_i, with high probability, the **Refine** query $\mathcal{Q}_i(p_{\mathcal{S}})$ is feasible. ■

Let the *selectivity* of a query be the probability of a random package being *infeasible*. Thus, the lower the selectivity of $\mathcal{Q}$, the higher the probability $Pr[\mathsf{P} \models \mathcal{Q}]$. Therefore, a consequence of Theorem 3 is that the lower the selectivity of $\mathcal{Q}$, the higher the probability that $\mathcal{Q}(\tilde{\mathsf{R}})$ and all $\mathcal{Q}_i(p_{\mathcal{S}})$ are feasible, which implies that **SketchRefine** will eventually find a feasible package with high probability as well.

3.4 Experimental evaluation of SketchRefine

In this section, we present an extensive experimental evaluation of our techniques for package query execution, both on real-world and on benchmark data. Our results show the following properties of our methods: (1) **SketchRefine** evaluates package queries an order of magnitude faster than **Direct**; (2) **SketchRefine** scales up to sizes that **Direct** cannot handle directly; (3) **SketchRefine** produces packages of high quality (similar objective value as the packages returned by **Direct**); (4) the performance of **SketchRefine** is robust to partitioning on different sets of attributes as long as a query's attributes are mostly covered. This makes offline partitioning effective for entire query workloads.

3.4.1 Experimental setup

Software. We implemented our package evaluation system as a layer on top of a traditional relational DBMS. The data itself resides in the database, and the system interacts with the DBMS via SQL when it needs to perform operations on the data. We use PostgreSQL v9.3.9 for our experiments. The core components of our evaluation module are implemented in Python 2.7. The PaQL parser is generated in C++ from a context-free grammar, using GNU Bison [66]. We represent a package in the relational model as a standard relation with schema equivalent to the schema of the input relation. A package is materialized into the DBMS only when necessary (for example, to compute its objective value).

We employ IBM's CPLEX [76] v12.6.1 as our black-box ILP solver. When the algorithm needs to solve an ILP problem, the corresponding data is retrieved from the DBMS and passed to CPLEX using tuple iterator APIs to avoid having more than one copy of the same data stored in main memory at any time. We used the same settings for all solver

TPC-H query	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Max # of tuples	6M	6M	6M	6M	240k	11.8M	6M

Table 3.1: Size of the tables used in the TPC-H benchmark.

executions: we set its working memory to 512MB; we instructed CPLEX to store exceeding data used during the solve procedure on disk in a compressed format, rather than using the operating system's virtual memory, which, as per the documentation, may degrade the solver's performance; we instructed CPLEX to emphasize optimality versus feasibility to dampen the effect of internal heuristics that the solver may employ on particularly hard problems; we enabled CPLEX's memory emphasis parameter, which instructs the solver to conserve memory where possible; we set a solving time limit of one hour; we also made sure that the operating system would kill the solver process whenever it uses the entire available main memory. Our code is publicly available on our project website: http://packagebuilder.cs.umass.edu.

Environment. We run all experiments on a ProLiant DL160 G6 server equipped with two twelve-core Intel Xeon X5650 CPUs at 2.66GHz each, with 15GB or RAM, with a single 7200 RPM 500GB hard drive, running CentOS release 6.5.

Datasets and queries. We demonstrate the performance of our query evaluation methods using both real-world and benchmark data. The real-world dataset consists of approximately 5.5 million tuples extracted from the Galaxy view of the Sloan Digital Sky Survey (SDSS) [151], data release 12. For the benchmark datasets we used TPC-H [153], with table sizes up to 11.8 million tuples.

We constructed a workload of seven feasible package queries for each dataset, by adapting existing SQL queries originally designed for each of the two datasets. For the Galaxy dataset,

Dataset	Dataset size	Size threshold τ	Partitioning time
Galaxy	5.5M tuples	550k tuples	348 sec.
TPC-H	17.5M tuples	1.8M tuples	1672 sec.

Table 3.2: Partitioning time for the two datasets, using the workload attributes and with no diameter condition.

we adapted real-world sample SQL queries available directly from the SDSS website.[1] For the TPC-H dataset, we adapted seven SQL query templates provided with the benchmark that contained enough numerical attributes. We performed query specification manually, by transforming SQL aggregates into global predicates or objective criteria whenever possible, selection predicates into global predicates, and by adding cardinality bounds. We did not include any base predicates in our package queries because they can always be pre-processed by running a standard SQL query over the input dataset (Section 2.4), and thus eliminated beforehand. For the Galaxy queries, we synthesized the global constraint bounds by multiplying the original selection bounds by the package cardinality bounds. For the TPC-H queries, we generated global constraint bounds uniformly at random by multiplying random values in the value range of a specific attribute by the cardinality bounds. We transformed the original TPC-H SQL queries into single-relation package queries by joining the original TPC-H tables using full outer joins, containing all attributes needed by all the TPC-H package queries in our benchmark. This pre-joined table contained approximately 17.5 million tuples. For each TPC-H package query, we then extracted the subset of tuples having non-NULL values on all the query attributes. The size of each resulting table is reported in Table 3.1. Finally, we do not allow tuple repetitions in any of the queries as they only affect the domains of the ILP integer variables. We observed that allowing tuple repetitions results in easier problems for the ILP solver.

[1] http://cas.sdss.org/dr12/en/help/docs/realquery.aspx

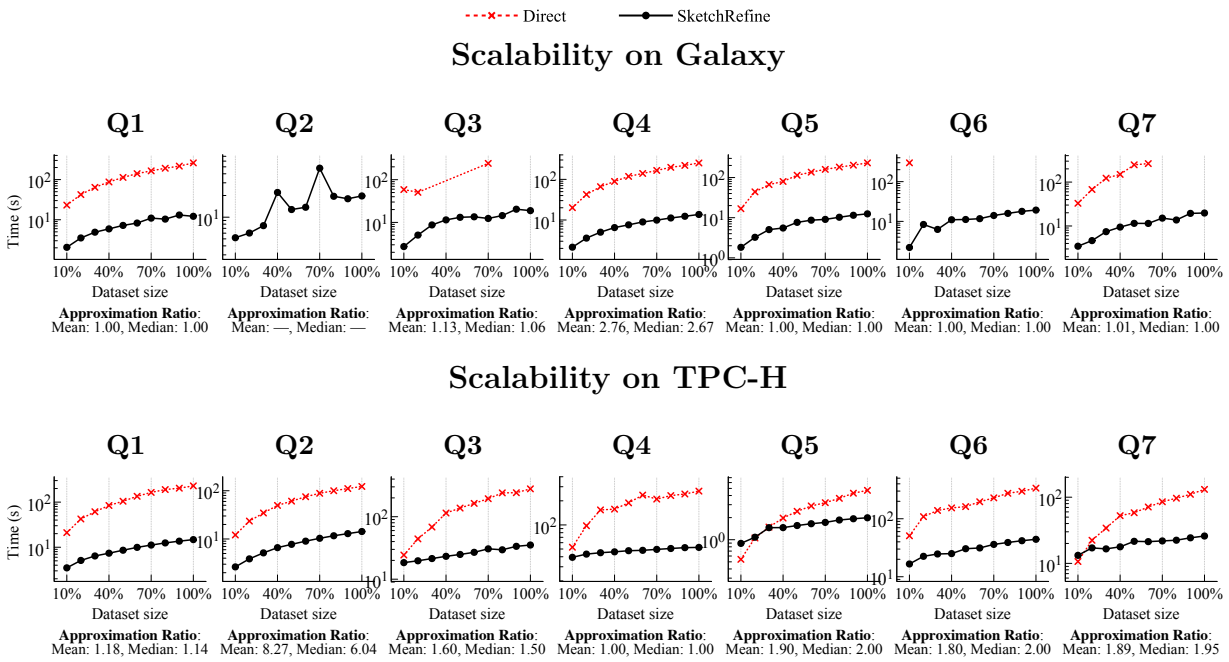


Figure 3.2: Scalability on the Galaxy and TPC-H benchmarks. SketchRefine uses an offline partitioning computed on the full dataset, using the workload attributes, $\tau = 10\%$ of the dataset size, and no diameter condition. In Galaxy, Direct scales up to millions of tuples in about half of the queries, but it fails on the other half. In TPC-H, Direct scales up to millions of tuples in all queries. SketchRefine scales up nicely in all cases, and runs about an order of magnitude faster than Direct. Its approximation ratio is generally very low, even though the partitioning is constructed without diameter conditions.

Comparisons. We compare **Direct** with **SketchRefine**. Both methods use the ILP formulation (Section 2.4) to transform package queries into ILP problems: **Direct** translates and solves the original query; **SketchRefine** translates and solves the subqueries (Section 3.2), and uses *hybrid sketch query* (Section 3.2.2.2) as the only strategy to cope with infeasible initial queries.

Metrics. We evaluate methods on their efficiency and effectiveness.

Response time: We measure response time as wall-clock time to generate an answer package. This includes the time taken to translate the PaQL query into one or several ILP problems,

the time taken to load the problems into the solver, and the time taken by the solver to produce a solution. We exclude the time to materialize the package solution to the database and to compute its objective value.

Approximation ratio: Recall that **SketchRefine** is always guaranteed to return an approximate answer with respect to **Direct** (Section 3.3.2). In order to assess the quality of a package returned by **SketchRefine**, we compare its objective value with the objective value of the package returned by **Direct** on the same query. Using Obj_S and Obj_D to denote the objective values of **SketchRefine** and **Direct**, respectively, we compute the empirical *approximation ratio* $\frac{Obj_D}{Obj_S}$ for maximization queries, and $\frac{Obj_S}{Obj_D}$ for minimization queries. An approximation ratio of one indicates that **SketchRefine** produces a solution with same objective value as the solution produced by the solver on the entire problem. Typically, the approximation ratio is greater than or equal to one. However, since the solver employs several approximations and heuristics, values lower than one, which means that **SketchRefine** produces a better package than **Direct**, are possible in practice.

3.4.2 Results and discussion

We evaluate four fundamental aspects of our algorithms: (1) their query response time and approximation ratio with increasing dataset sizes; (2) the impact of varying partitioning size thresholds, τ, on **SketchRefine**'s performance; (3) the impact of the attributes used in offline partitioning on query runtime; (4) the impact of enforcing approximation guarantees, ϵ, on the performance of **SketchRefine**.

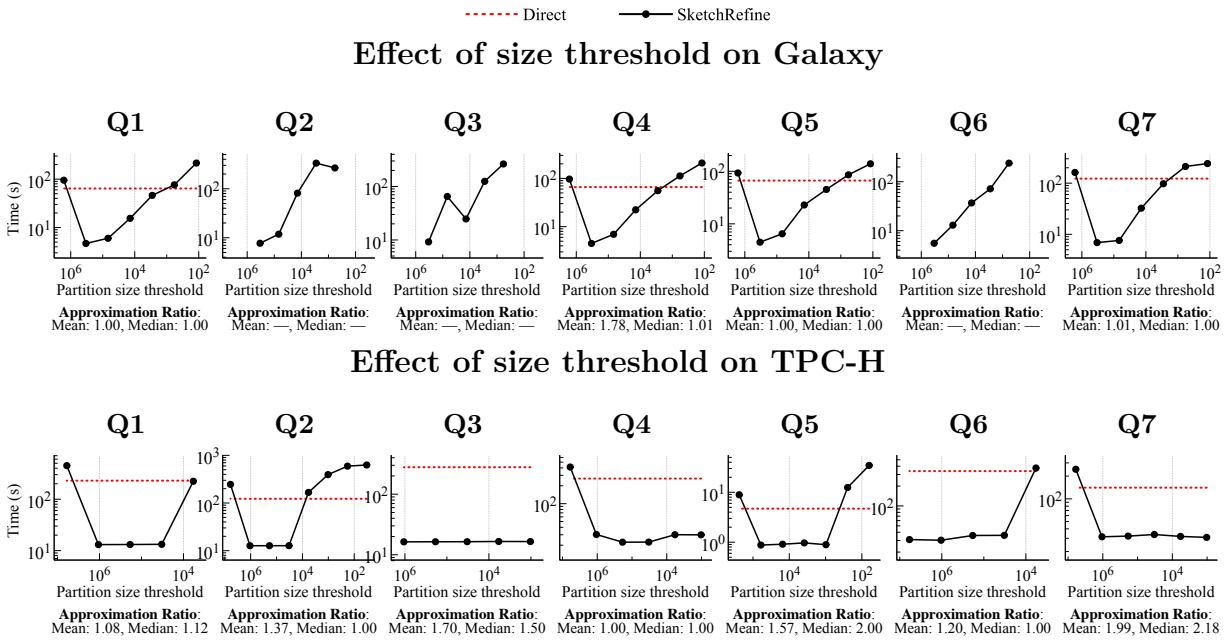


Figure 3.3: Impact of partition size threshold τ on the Galaxy and TPC-H benchmarks, using, respectively, 30% and 100% of the dataset. Partitioning is performed at each value of τ using all the workload attributes, and with no diameter condition. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. The results show that τ has a major impact on the running time of SKETCHREFINE, but almost no impact on the approximation ratio. SKETCHREFINE can be an order of magnitude faster than DIRECT with proper tuning of τ.

3.4.2.1 Query performance as dataset size increases

In our first set of experiments, we evaluate the scalability of our methods on input relations of increasing size. First, we partitioned each dataset using the union of all package query attributes in the workload: we refer to these partitioning attributes as the *workload attributes*. We did not enforce diameter conditions, ω_{ij}, during partitioning for three reasons: (1) because the diameter conditions may affect the size of the resulting partitions, and we want to tightly control the partition size through the parameter τ; (2) to show that an offline partitioning can be used to answer efficiently and effectively both maximization and minimization queries, even though they would normally require different diameters; (3) to demonstrate the effectiveness of **SketchRefine** in practice, even without having theoretical guarantees in place. Because we do not enforce approximation guarantees, the group centroids are used as representatives for all queries. In Section 3.4.2.4, we specifically test how varying the diameter requirements through ϵ affects the running time of **SketchRefine**.

We perform offline partitioning setting the partition size threshold τ to 10% of the dataset size. Table 3.2 reports the partitioning times for the two datasets. We derive the partitionings for the smaller data sizes (less than 100% of the dataset) in the experiments, by randomly removing tuples from the original partitions. This operation is guaranteed to maintain the size condition.

Figure 3.2 reports our scalability results on the Galaxy and TPC-H benchmarks. The figure displays the query response time in seconds on a logarithmic scale, averaged across 10 runs for each datapoint. At the bottom of each plot, we also report the mean and median approximation ratios across all dataset sizes. The graph for Q2 on the galaxy dataset does not report approximation ratios, because **Direct** evaluation fails to produce a solution for this query across all data sizes. We observe that **Direct** can scale up to

millions of tuples in three of the seven Galaxy queries, and in all of the TPC-H queries. Its run-time performance degrades, as expected, when data size increases, but even for very large datasets **Direct** is usually able to answer the package queries in less than a few minutes. However, **Direct** has high failure rate for some of the Galaxy queries, indicated by the missing data points in some graphs (queries Q2, Q3, Q6 and Q7 in the Galaxy dataset). This happens when CPLEX uses the entire available main memory while solving the corresponding ILP problems. For some queries, such as Q3 and Q7, this occurs with bigger dataset sizes. However, for queries Q2 and Q6, **Direct** even fails on small data. This is a clear demonstration of one of the major limitations of ILP solvers: they can fail even when the dataset can fit in main memory, due to the complexity of the integer problem. In contrast, our scalable **SketchRefine** algorithm is able to perform well on all dataset sizes and across all queries. **SketchRefine** consistently performs about an order of magnitude faster than **Direct** across all queries, both on real-world data and benchmark data. Its running time is consistently below one or two minutes, even when constructing packages from millions of tuples.

Both the mean and median approximation ratios are very low, usually all close to one or two. This shows that the substantial gain in running time of **SketchRefine** over **Direct** does not compromise the quality of the resulting packages. Our results indicate that the overhead of partitioning with diameter limits is often unnecessary in practice. Since the approximation ratio is not enforced, **SketchRefine** can potentially produce bad solutions, but this happens rarely. In our experiments, this only occurred with query Q2 from the TPC-H benchmark.

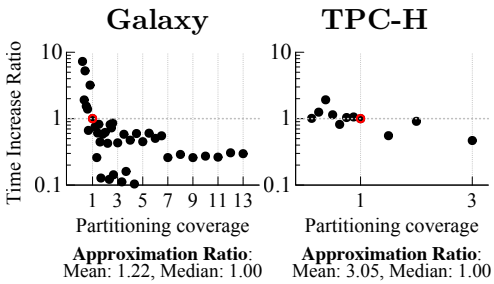


Figure 3.4: Increase or decrease ratio in running time of SketchRefine with different partitioning coverages. Coverage one, shown by the red dot, is obtained by partitioning on the query attributes. The results show an improvement in running time when partitioning is performed on supersets of the query attributes, with very good approximation ratios.

3.4.2.2 Effect of varying partition size threshold

The size of each partition, controlled by the partition size threshold τ, is an important factor that can impact the performance of **SketchRefine**: Larger partitions imply fewer but larger subproblems, and smaller partitions imply more but smaller subproblems. Both cases can significantly impact the performance of **SketchRefine**. In our second set of experiments, we vary τ, which is used during partitioning to enforce the size condition (Section 3.2.1), to study its effects on the query response time and the approximation ratio of **SketchRefine**. In all cases, along the lines of the previous experiments, we do not enforce diameter conditions and pick each group's centroid as the representative. Figure 3.3 shows the results obtained on the Galaxy and TPC-H benchmarks, using 30% and 100% of the original data, respectively. We vary τ from higher values corresponding to fewer but larger partitions, on the left-hand size of the x-axis, to lower values, corresponding to more but smaller partitions. When **Direct** is able to produce a solution, we also report its running time (horizontal line) as a baseline for comparison.

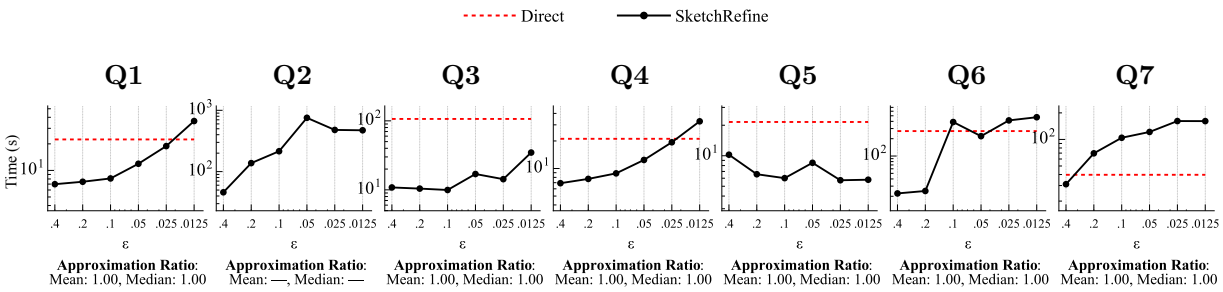


Figure 3.5: Impact of the approximation parameter ϵ (increasingly stricter approximation requirements) on the Galaxy workload, using 10% of the dataset size. Partitioning is performed on the query attributes, without enforcing a limit on the size of the partitions, τ, while imposing diameter limits governed by ϵ. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. The results show that ϵ has a major impact on the running time of SKETCHREFINE, as a smaller ϵ implies smaller partition diameters and, thus, more partitions, while maintaining the approximation ratio always down to 1.

Our results show that the partition size threshold has a major impact on the execution time of **SKETCHREFINE**, with extreme values of τ (either too low or too high) often resulting in slower running times than **DIRECT**. With bigger partitions, on the left-hand side of the x-axis, **SKETCHREFINE** takes about the same time as **DIRECT** because both algorithms solve problems of comparable size. When the size of each partition starts to decrease, moving from left to right on the x-axis, the response time of **SKETCHREFINE** decreases rapidly, reaching about an order of magnitude improvement with respect to **DIRECT**. Most of the queries show that there is a "sweet spot" at which the response time is the lowest: when all partitions are small, and there are not too many of them. The point is consistent across different queries, showing that it only depends on the input data size (refer to Table 3.1 for the different TPC-H data sizes). After that point, although the partitions become smaller, the number of partitions starts to increase significantly. This increase has two negative effects: it increases the number of representative tuples, and thus the size and complexity of the initial **SKETCH** query, and it increases the number of groups

that **Refine** may need to refine to construct the final package. This causes the running time of **SketchRefine**, on the right-hand side of the x-axis, to increase again and reach or surpass the running time of **Direct**. We only report mean and median approximation ratios, which are in all cases very close to one, indicating that **SketchRefine** retains very good quality regardless of the partition size threshold. We studied how different partitioning size thresholds (τ) affect approximation ratios. We observed that the ratio follows an inverse trend to that of the running time in Figure 3.3. In the two extreme cases, when there is only one partition of size n (**SketchRefine** is a single refine query that corresponds to **Direct**) and when there are n partitions of size 1 (**SketchRefine** is a sketch query over n groups of a single tuple each), **SketchRefine** returns the optimal solution (approximation ratio 1). Between these endpoints, for some queries, the approximation ratio can be higher than 1. With a smaller number of partitions, our partitioning algorithm produces larger partitions with potentially large diameters, but each refine query produces an optimal solution over a larger subproblem. As the number of partitions increases, the refine query operates over smaller subproblems leading to worse approximation ratios, until the partitions start to have tighter diameters leading to better approximation.

3.4.2.3 Effect of varying partitioning coverage

In this experiment, we study the impact of offline partitioning on the query response time and the approximation ratio of **SketchRefine**. We define the *partitioning coverage* as the ratio between the number of partitioning attributes and the number of query attributes. For each query, we test partitionings created using: (a) exactly the query attributes (coverage $= 1$), (b) proper subsets of the query attributes (coverage < 1), and (c) proper supersets of the query attributes (coverage > 1).

For each query, we report the effect of the partitioning coverage on query runtime as the ratio of a query response time over the same query's response time when coverage is one: a higher ratio (> 1) indicates slower response time and a lower ratio (< 1) indicates a faster response time. Figure 3.4 reports the results on the Galaxy and the TPC-H datasets. The Galaxy dataset has many more numerical attributes than the TPC-H dataset, allowing us to experiment with higher values of coverage. The response time of **SketchRefine** improves on both datasets when the offline partitioning covers a superset of the query attributes, whereas it tends to increase when it only considers a subset of the query attributes. The mean and median approximation ratios are consistently low, indicating that the quality of the packages returned by **SketchRefine** remains unaffected by the partitioning coverage.

These results demonstrate that **SketchRefine** is robust to imperfect partitioning, which do not cater precisely to the query attributes. Moreover, using a partitioning over a superset of a query's attributes typically leads to better performance. The reason for this is twofold: First, higher coverage achieves partitioning groups where tuples are similar across all attributes pertinent to the query. Thus, the sketch query uses better representatives and produces a more relevant initial package, and the refine queries are more likely feasible. Second, partitioning on more attributes can also achieve smaller partitioning groups. As a result, this speeds up the refine queries, and also reduces the diameter of each group, with the potential of improving the approximation ratio. This means that partitioning can be performed offline using the union of the attributes of an anticipated workload, or even using all the attributes of a relation.

3.4.2.4 Effect of varying ϵ

In our final set of experiments, we study the impact of different approximation guarantees on the query response time and the approximation ratio of **SketchRefine**. We vary ϵ, the approximation parameter, from higher values (looser approximation bound) to lower values (tighter approximation bound), and enforce diameter limits according to Theorem 2. A looser approximation bound can cause the algorithm to produce package results with a worse objective value. More specifically, $\epsilon = 0.4$ guarantees approximation ratios not worse than 1.4 for minimization queries and 1.67 for maximization queries, and $\epsilon = 0.0125$ guarantees approximation ratios not worse than 1.0125 for minimization queries and 1.0127 for maximization queries. Figure 3.5 presents the results on the Galaxy workload, where ϵ varies from high values, on the left-hand size of the x-axis, to lower values. When **Direct** is able to produce a solution, we also report its running time (horizontal line) as a baseline for comparison.

Enforcing stricter (lower) ϵ leads to an increase in the running time of **SketchRefine**. This is expected, as the stricter diameter bounds result in more partitions, and as we observed in our partition threshold experiments (Section 3.4.2.2), having more partitions can negatively impact the running time of **SketchRefine**. This trade-off between quality and runtime performance is a known characteristic of most approximation schemes [159].

Our results also show that enforcing even a loose ϵ, such as 0.4, enables **SketchRefine** to compute a result to all the queries faster than **Direct** with no cost in quality, as the observed approximation ratios are always equal to 1. Notably, this happens in all the queries, including those that showed higher approximation ratio in the previous experiments where the approximation guarantee was not enforced.

3.5 Parallelizing SketchRefine

Our evaluation showed that **SketchRefine** outperforms **Direct** on both the Galaxy and the TPC-H datasets. Specifically, **SketchRefine** has three important advantages: First, it scales naturally to very large datasets, by breaking down the problem into smaller, manageable subproblems, whose solutions can be combined to form the final result. Second, it provides flexible approximations with strong theoretical guarantees on the quality of the package results. Third, while our current implementation employs ILP solvers, **SketchRefine** can use any arbitrary black-box algorithm to evaluate the generated package subproblems, even solutions that work entirely in main memory [67, 159, 63], and whose efficiency drastically degrades with larger problem sizes. **SketchRefine** will offer the same efficiency gains and approximation guarantees over the employed black-box algorithm.

However, there are two scenarios that can degrade **SketchRefine**'s performance. First, as we discussed in Section 3.2.2.2, the worst-case running time of the algorithm is exponential in the number of partitions, due to the backtracking logic in the **Refine** phase. The **Refine** algorithm may get caught in a sequence of promising refine orderings that fail at their last step. Our evaluation showed that this scenario is uncommon in practice, and the algorithm was always able to quickly find a successful refine order for the partitioning groups. Second, **SketchRefine** achieves most of its gains in the **Sketch** phase, which identifies the relevant partitions, reducing the work of **Refine**. Thus, the algorithm is susceptible to bad performance when queries require tuples to be picked from a large number of the partitions. We investigate this scenario in more detail, starting with a motivating example from the Galaxy dataset.

Example 7 (Varied Red Galaxies)**.** Similar to Example 2, an astrophysicist is looking for rectangular regions of the night sky that may contain previously unseen celestial objects.

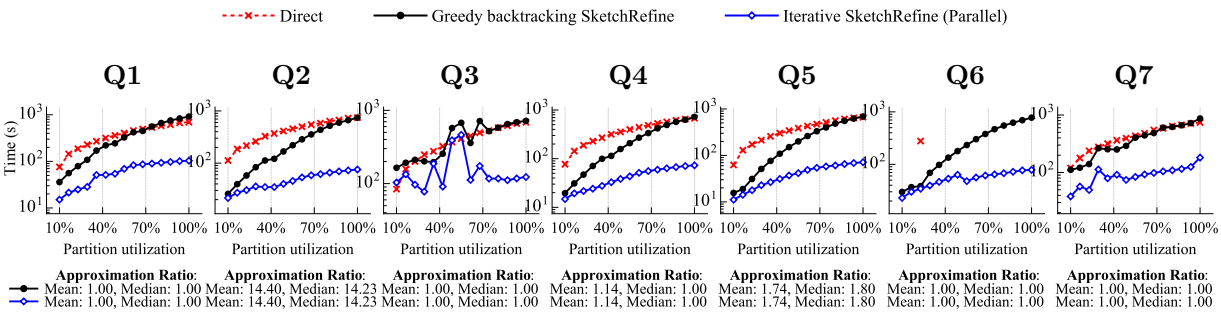


Figure 3.6: Impact of increased partition utilization on SketchRefine, on 30% of the Galaxy data. Partitioning is over attribute r only, using $\tau = 10\%$ and no diameter bounds. The performance of SketchRefine degrades as partition utilization increases, approaching the runtime of Direct. The runtime of Direct also increases, as the constraints that force higher partition utilization increase the complexity of the query.

This time, the scientist is specifically looking for galaxies that span different brightness levels on the red color component.

In this example, the astrophysicist requires each galaxy (package) to include red color components from the entire red spectrum. We can encode this in PaQL by dividing the red spectrum into ranges, and requiring the resulting package to include at least one tuple from each range interval. Each such constraint would be of the following form:

(SELECT COUNT(*) FROM P
WHERE r BETWEEN r_{lb} AND r_{ub}) >= 1

where r is the name of the red color component from the Galaxy schema, and r_{lb} and r_{ub} are the lower and upper bounds of one of the range intervals. The query has one such constraint for each range interval. Each such constraint forces the result package to contain at least one tuple in the specified r range.

If the dataset is partitioned on the red color component, r, these constraints will force **SketchRefine** to generate and solve a subproblem for most of the partitions, causing a

substantial increase to its running time. In the worst case, **Refine** will need to operate on all partition groups, and its performance can get as bad as **Direct**.

We implement the scenario of this example in our Galaxy workload by partitioning the data on attribute r, generating 14 partitioning groups. We create constraints based on range intervals that correspond to the partitioning on r. Then for each Galaxy query, we generate a sequence of 14 queries, by augmenting the query with more constraints on r, thus forcing increasingly higher partition utilization. The first query of the sequence only has one color constraint requiring at least one tuple from a single partition, corresponding to the lowest partition utilization ($\sim$10%). The last query of the sequence has 14 color constraints, one for each partition, requiring at least one tuple from *each* partition. This corresponds to the highest partition utilization (100%). Queries with more constraints on r will require the **Refine** phase to solve more partitions. We observe the impact of this workload on **SketchRefine**'s performance in Figure 3.6: As partition utilization increases (due to more constraints on r), the runtime of greedy **SketchRefine** increases, and matches that of **Direct** when most partitions are needed. In this experiment, the runtime of **Direct** also increases, as the addition of the partition constraints makes each query individually more complex.

Since **SketchRefine** relies on solving several smaller subproblems, a natural way to improve its performance is by parallelizing the **Refine** step. Unfortunately, the greedy backtracking algorithm (Algorithm 2) requires incremental refinements, always maintaining the feasibility of the intermediate solutions. Each step in the algorithm makes a local decision based on results of the previous decisions and their order. Thus, solving the **Refine** subproblems in parallel does not guarantee that the overall package will be feasible.

In this section, we introduce a new *iterative* method for performing the **Refine** phase of **SketchRefine**. The iterative algorithm has the following advantages over Algorithm 2:

(1) It allows partitions to be evaluated in parallel, independently from each other; (2) It eliminates the need for backtracking and, thus, its exponential worst-case; (3) It can reach infeasibility faster than backtracking, while offering the same false-infeasibility bounds; (4) It guarantees the same approximation bounds. Figure 3.6 shows that parallel execution of our new *iterative* **SketchRefine** leads to significant gains in performance, and avoids the degradation that greedy **SketchRefine** demonstrates in cases of high partition utilization. We proceed to describe the new **Refine** algorithm, explain how it parallelizes, and demonstrate its scalability.

3.5.1 Iterative Refine

The **Refine** step of **SketchRefine** processes the sketch package p_S to replace the representative tuples with tuples from each partition. It does this by defining and solving appropriate ILP problems within each partition (refinement). The greedy backtracking implementation of **Refine** (Algorithm 2) performs refinements one at a time, and requires each refinement to yield a feasible package for the original query; if a refinement does not, the algorithm backtracks. We now present an alternative strategy for the **Refine** step that relaxes this requirement until a tuple solution for every partitioning group is found. Specifically, *iterative* **Refine** performs refinements independently on each partition, modifying the sketch package based on all the successful refinements, and repeating any failed ones using the new revised sketch package. Only after all partitions are solved, the algorithm ensures feasibility of the resulting package. Algorithm 3 details the procedure, which works in two phases:

Phase 1: Iterative refinements. The first phase of the algorithm (lines 2–19) performs refinements on all unsolved partitions ($\mathcal{S}$) iteratively. At each iteration, for each unsolved partition, the algorithm solves an ILP (constructed as in Section 3.2.2.2) to replace the representative tuples with tuples from the partition. The algorithm updates the sketch

package ($p_{\mathcal{S}}$) based on the refinements (line 15). This process repeats while there are still unsolved partitions, i.e., partitions that failed to produce a feasible solution in previous iterations (line 3). The refinement queries $\mathcal{Q}_i(p_{\mathcal{S}})$ in the new iterations will be different from their earlier versions, as the constraints on each refine query depend on $p_{\mathcal{S}}$, which has been modified by the previous iterations. Phase 1 fails (line 19) if, during an iteration, none of the partition groups can be solved: In this case, $\mathcal{S}$ remains unchanged, and the algorithm cannot make progress towards the completion of the package.

During this phase, the algorithm does not check whether $p_{\mathcal{S}}$ is a feasible solution to the overall query. Rather, the objective of this phase is to produce feasible solutions for *each* of the partition groups.

Phase 2: Feasibility adjustment. If Phase 1 concludes successfully, the algorithm enters Phase 2 (lines 20–29) to verify whether $p_{\mathcal{S}}$ is a feasible solution to the overall query and attempt a correction if it is not. If $p_{\mathcal{S}}$ is not a feasible solution, the algorithm tries one more refine round of all partitions based on the current $p_{\mathcal{S}}$. If any of the refine queries succeeds in this round, then the new, refined $p_{\mathcal{S}}$ is guaranteed to be a feasible solution and the algorithm returns it. If all refinement queries are infeasible, the algorithm fails. Thus, iterative **Refine** may fail in two cases: (1) if all refining queries fail in one iteration of Phase 1; or (2) if the refined sketch package $p_{\mathcal{S}}$ is infeasible and unfixable in Phase 2.

Run time complexity. We denote with $T(\tau)$ the time taken by the solver to solve a problem of size τ, and express the time complexity of the refine procedure as a function of $T(\tau)$ and m, the number of partitioning groups. The best case for Algorithm 3 is that all refine queries succeed in the first iteration of Phase 1 and, in Phase 2, the refined $p_{\mathcal{S}}$ is already a feasible solution. In this case, the algorithm makes up to m calls to the solver. In the worst case, only one refine query succeeds in each iteration of Phase 1, the refined

package p_S is not a feasible solution to the overall query, and only the last attempt of Phase 2 succeeds in rendering p_S feasible. In this case, Algorithm 3 makes up to $\frac{m(m+1)}{2} + m$ calls to the solver ($O(T(\tau)m^2)$).

Comparison with greedy backtracking Refine. However, in sequential settings greedy backtracking can outperform iterative **Refine** in practice. Specifically, if the subproblems can be solved independently of each other, but fail when combined, iterative **Refine** requires extra steps in Phase 2 to adjust the solution. On the other hand, greedy backtracking would terminate as soon as all subproblems are solved, as it always maintains feasible solutions. With infeasible groups, iterative **Refine** may also require several Phase 1 iterations, while greedy backtracking would immediately backtrack at the first infeasible group. This means that Algorithm 2 is likely to beat Algorithm 3 in harder problems, which have few feasible solutions.

3.5.2 Parallelizing iterative Refine

Iterative **Refine** is naturally amenable to parallelization, since all refinement problems are solved independently from each other. In particular, during Phase 1, the algorithm solves groups independently without ensuring the feasibility of the overall package. Therefore, all the refine queries in each iteration of Phase 1 can be solved in parallel, and their solutions can be combined by a central node at the end of every iteration. During Phase 2, all the refine queries are also independent because the algorithm can stop if *any* of them succeeds. Thus, all refine queries of Phase 2 can also be executed in parallel, and if any succeeds, the other ones can be immediately terminated. A central node dispatches the refine queries to be solved at each iteration to the parallel worker nodes, and combines their result into p_S, the refining sketch package. Thus, every worker node is responsible for a different

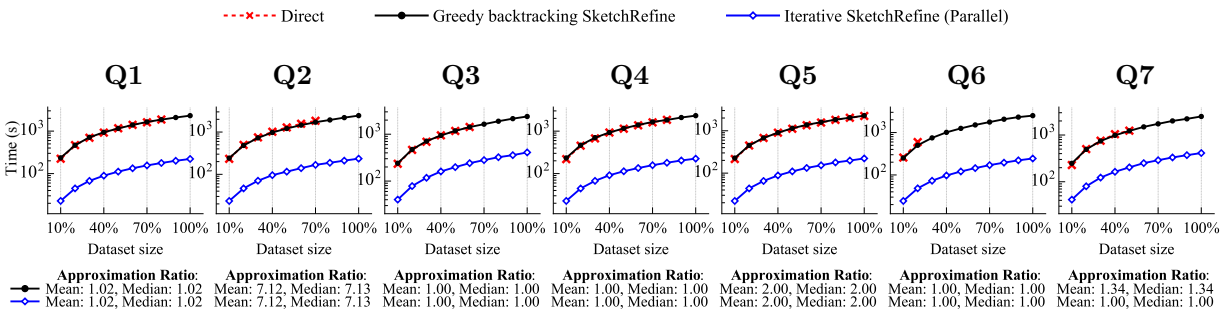


Figure 3.7: Scalability of parallel SketchRefine compared to greedy backtracking SketchRefine and Direct on the *varied red* Galaxy workload. SketchRefine uses partitioning computed on attribute r, $\tau = 10\%$, and no diameter condition. The running time of greedy backtracking SketchRefine and Direct are equal as all partitions need to be refined (worst case of greedy backtracking). Parallel SketchRefine scales up to nicely in all cases, and runs about an order of magnitude faster than both Direct and greedy backtracking. The approximation ratios of the two algorithms are both generally low, even though the partitioning is constructed without quality guarantees in place.

partitioning group. If there are more partitioning groups than workers, the load can be easily balanced among the workers by assigning them to an equal number of groups.

3.5.3 Experimental evaluation of parallel SketchRefine

We evaluate the scalability and effectiveness of parallel **SketchRefine** using a variation of the queries of our Galaxy workload based on Example 7. Specifically, we partition our data on the red color component attribute, r, with $\tau = 10\%$ of the original dataset size and no diameter conditions, and we modify the Galaxy queries to include cardinality constraints on ranges of r. Our partitioning on r generates 14 groups, and the runtime improvements that we report in this section are achievable with 14 parallel worker nodes (one for each partitioning group). In each experiment, we measure the running time and the approximation ratio (described in Section 3.4.1) of the algorithms for increasing dataset

sizes, comparing **Direct** with two versions of **SketchRefine**: one that uses greedy backtracking **Refine** (Algorithm 2), and one that uses iterative **Refine** (Algorithm 3).

In our first experiment, we change the number of cardinality constraints on ranges of r for each query: the more constraints, the more partitions **SketchRefine** will need to explore. As we have seen, the performance of **SketchRefine** with greedy backtracking degrades as partition utilization increases (Figure 3.6). In contrast, we observe that parallel iterative **SketchRefine** maintains consistently better performance than **Direct**.

For our second experiment, we pick the query workload with the highest partition utilization (100%), which requires *all* of the partitions to be refined. Figure 3.7 reports the results. All queries show similar performance because they all share the same 14 cardinality constraints on r. Both of the **SketchRefine** versions scale to millions of tuples, whereas **Direct** fails in many of the queries when the dataset gets too big. Here, **Direct** fails for the same reasons as our earlier experiments in Section 3.4.2.1. In all the cases in which **Direct** succeeds, as the dataset size increases, greedy backtracking **SketchRefine** shows the same run-time performance as **Direct**. In fact, requiring galaxies that span all of the red color ranges requires tuples to be picked from each partition, which corresponds to the worst case for greedy backtracking. On the other hand, parallel **SketchRefine** is able to always find an answer in about an order of magnitude less time than greedy backtracking and **Direct**. This happens because the algorithm is able to parallelize all the necessary refinements.

In this set of experiments, we did not enforce approximation guarantees, so the algorithms can potentially produce bad solutions. However, our results show that this happens rarely, and the approximation ratios of both of the **SketchRefine** algorithms are generally very low (close to one). One exception is query Q2, for which **SketchRefine** produces a 7-factor approximation. Finally, the approximation quality of parallel iterative

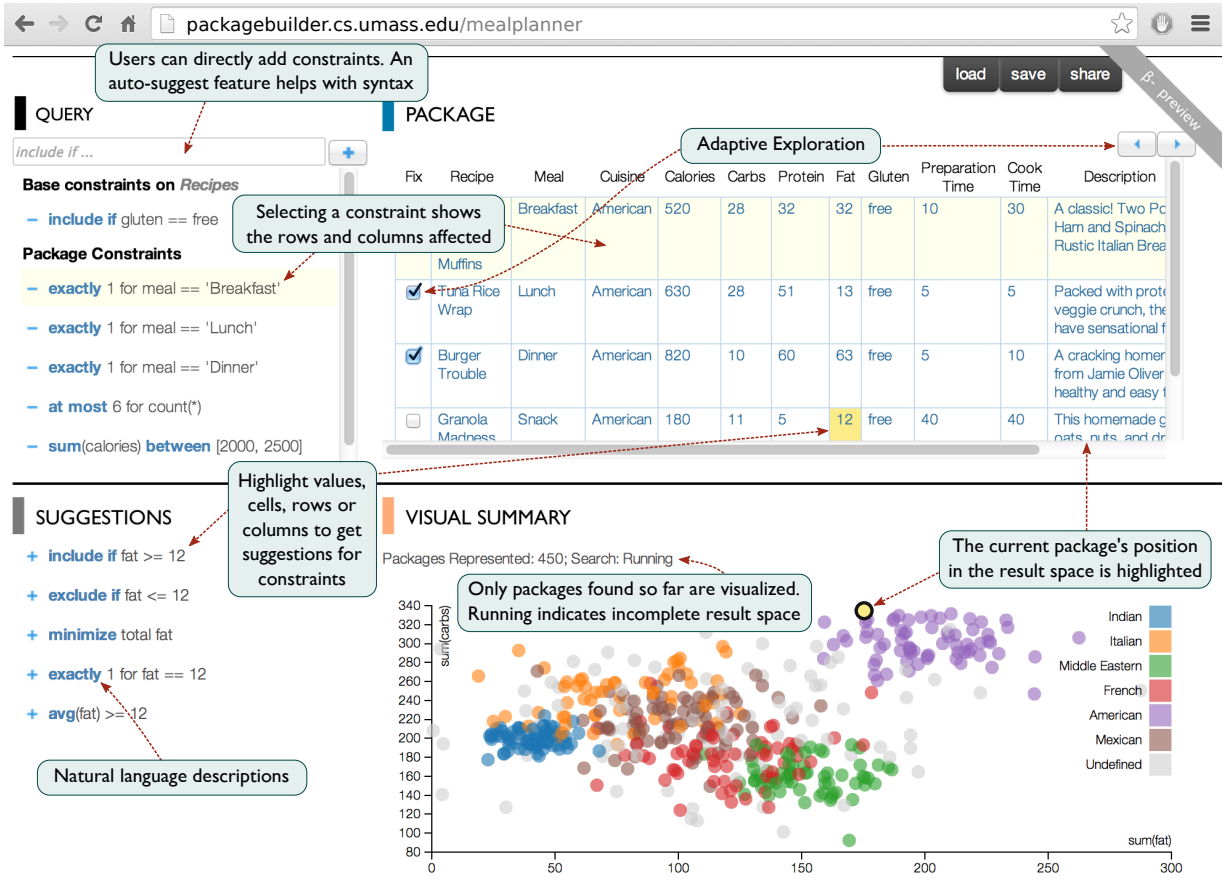


Figure 3.8: The visual interface of PackageBuilder provides different visual representations of packages, and allows the user to interactively manipulate package queries.

SketchRefine is equal (queries Q1–Q6) or better (Q7) than greedy backtracking. This shows that the gains obtained by parallelizing **SketchRefine** do not come at the cost of quality and, in some cases, can also produce better solutions.

3.6 An interface for querying and manipulating packages

Package queries are more complex, semantically and algorithmically, compared to traditional database queries, and they pose challenges on several fronts: they can have complex specifications, and they produce a large number of results, which poses usability

challenges [27]. In this section, we describe several interface abstractions that PackageBuilder implements to address these challenges.

3.6.1 Specification

Our *package template* abstraction encodes package specifications in a familiar tabular format (Figure 3.8 shows a screenshot example). The central component of the template is a *sample package*, presented as a scrollable table. Additional components include representations of base and global constraints, optimization objectives, and suggestions for additional package refinements. As a user interacts with the template by highlighting elements in the sample package, PackageBuilder suggests constraints [86, 6]. For example, when the user selects a cell within the "fats" column, the system proposes several constraints that would restrict the amount of fat in each meal, and objectives that would minimize the total amount of fat. The package template is quite expressive but is not as powerful as the PaQL language itself. The abstraction tries to strike a balance between ease-of-use and expressive power.

3.6.2 Presentation

In addition, PackageBuilder presents packages in a way that allows users to meaningfully view the entire package space, without having to actually examine it in its entirety (see the *visual summary* at the bottom of Figure 3.8). The system analyzes the current query specification and selects two dimensions to visually layout the valid packages along. Users can use the visual summary to navigate through the available packages by selecting glyphs that represent them.

3.6.3 Adaptive exploration

Many users may prefer specifying queries in trial-and-error, incremental form, rather than providing a complete and precise specification from the very beginning. To facilitate this approach, PackageBuilder initially presents a sample package that satisfies a few basic constraints. Users can then select good tuples within the sample, and request a new sample that replaces the unselected tuples. Users can repeat this process until they reach the ideal package. PackageBuilder uses these selections to narrow the search space as well as to identify additional package constraints.

3.6.4 Cardinality-based pruning

With pruning techniques, the system can avoid generating candidate packages that cannot possibly satisfy some of the global constraints. Given a global constraint $\mathcal{C}$, our pruning strategy identifies a lower cardinality bound l and an upper cardinality bound u for any package that can satisfy $\mathcal{C}$. For example, if $\mathcal{C}$ is defined as $a \leq \mathsf{COUNT}(*) \leq b$, the cardinality bounds are trivially $l = a$ and $u = b$. As another example, consider the global constraint on total calories per package: $2000 \leq \mathsf{SUM}(\mathsf{calories}) \leq 2500$. In this case, the cardinality bounds are $l = \lceil \frac{2000}{\mathsf{MAX}(\mathsf{calories})} \rceil$ and $u = \lfloor \frac{3000}{\mathsf{MIN}(\mathsf{calories})} \rfloor$. In fact, with at least l recipes with $\mathsf{MAX}(\mathsf{calories})$ and at most u recipes with $\mathsf{MIN}(\mathsf{calories})$ we can achieve both the lower and upper bounds of the summation constraint.

Assuming queries that do not allow repeated tuples, if n tuples satisfy the base constraints, this pruning approach reduces the search space from 2^n to $\binom{n}{l} + \binom{n}{l+1} + \cdots + \binom{n}{u-1} + \binom{n}{u}$, without losing any valid solution.

3.6.5 Heuristic local search

Pruning often reduces the search space significantly, but this reduction alone is seldom sufficient. In addition to pruning algorithms, which reduce the search space while maintaining completeness, PackageBuilder employs a heuristic local search to hasten the computation. As with any heuristic, there is no guarantee that all valid solutions will be found. Given a starting package P_0 (which can be constructed, for example, at random), PackageBuilder identifies all possible k-tuple replacements that can lead to a valid package, by using a single SQL query. For example, suppose we wish to generate meal packages with less than 2,500 total calories. Given a package P_0 having a total of 3,000 calories, we can identify all possible single-tuple replacements which lead to valid packages with the following SQL query:

SELECT P_0.id, R.id
FROM P_0, Recipes R
WHERE $3000 - P_0.\text{calories} + R.\text{calories} \leq 2500$

This query implements a greedy heuristic that is only able to locate valid packages that differ from P_0 by one single tuple. It fails to find any valid package that differ from P_0 by more than one tuple. The query can be also modified to explore packages of different cardinalities in a straightforward way. Notice that the query is a selection over a Cartesian product between the candidate package and the recipe relation. This approach is very efficient if we are attempting to replace only a few tuples at a time. For k replacements, however, this method would require a $2k$-way join, which quickly becomes intractable.

This local search is also particularly useful for adaptive exploration (Section 3.6.3), where users usually request the replacement of only a few tuples at a time.

3.6.6 Example usage scenario

The interface of PackageBuilder shown in Figure 3.8 was demonstrated [27] on a real-world application: the *meal planner*. Meal planner has a rich recipe data set scrapped from online recipe and nutrition websites. Users can observe how packages are specified with the package template, and interactively refined with adaptive exploration.

3.7 Conclusion

In this chapter, we introduced a complete system that supports the specification and efficient evaluation of package queries. We presented PaQL, a declarative extension to SQL, and theoretically established its expressiveness, and we developed a flexible approximation method, with strong theoretical guarantees, for the evaluation of PaQL queries on large-scale datasets. Our experiments on real-world and benchmark data demonstrate that our scalable evaluation strategy is effective and efficient over varied data sizes and query workloads, and remains robust under suboptimal conditions, parameter settings, and queries that require most of the partitions to be accessed at query time. We extended our **SketchRefine** method to allow for effective parallelization, and we demonstrated that it maintains good performance in adverse query scenarios. Limitations of **SketchRefine** and future research directions are discussed in Chapter 5, Section 5.1.

Algorithm 2 Greedy backtracking **Refine**

input:

- $\mathcal{Q}$: the package query to be evaluated
- $\mathcal{P} = \{(G_1, \tilde{t}_1), \ldots, (G_m, \tilde{t}_m)\}$: partitioning groups
- $\mathcal{S}$: partitioning groups yet to be refined (initially $\mathcal{S} = \mathcal{P}$)
- $p_{\mathcal{S}}$: the refining package (initially the result of **Sketch**)

output: a feasible package containing only tuples, or failure

procedure **Refine**($\mathcal{Q}$, $\mathcal{P}$, $p_{\mathcal{S}}$)
  $\mathcal{F} \leftarrow \emptyset$ ▷ Failed groups
  if $\mathcal{S} = \emptyset$ **then** ▷ Base case: all groups already refined
    return $(p_{\mathcal{S}}, \mathcal{F})$
  ▷ Arrange $\mathcal{S}$ in some initial order (e.g., random)
  $\mathcal{U} \leftarrow priorityQueue(\mathcal{S})$
  while $\mathcal{U} \neq \emptyset$ **do**
    $(G_i, \tilde{t}_i) \leftarrow dequeue(\mathcal{U})$
    ▷ Skip groups that have no representative in $p_{\mathcal{S}}$
    if $\tilde{t}_i \notin p_{\mathcal{S}}$ **then**
      continue
    $p_i \leftarrow$ **Direct**$(\mathcal{Q}_i(p_{\mathcal{S}}))$
    if $\mathcal{Q}_i(p_{\mathcal{S}})$ **is feasible then**
      ▷ Replace representative with tuples
      $p'_{\mathcal{S}'} \leftarrow p_{\mathcal{S}} \setminus \tilde{p}_i \cup p_i$
      $\mathcal{S}' \leftarrow \mathcal{S} \setminus \{(G_i, \tilde{t}_i)\}$
      ▷ Greedily *recurse* with refinable group
      $(p, \mathcal{F}') \leftarrow$ **Refine**$(\mathcal{Q}, \mathcal{P}, p'_{\mathcal{S}'})$
      if $\mathcal{F}' \neq \emptyset$ **then** ▷ **Refine** failure
        $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}'$
        ▷ Greedily *prioritize* non-refinable groups
        $prioritize(\mathcal{U}, \mathcal{F})$
      else ▷ **Refine** success
        return $(p, \mathcal{F})$
    else ▷ $\mathcal{Q}_i(p_{\mathcal{S}})$ is infeasible
      if $\mathcal{S} \neq \mathcal{P}$ **then** ▷ If $p_{\mathcal{S}}$ is not the initial package
        ▷ Greedily *backtrack* with non-refinable group
        $\mathcal{F} \leftarrow \mathcal{F} \cup \{(G_i, \tilde{t}_i)\}$
        return $(null, \mathcal{F})$
  ▷ None of the groups in $\mathcal{S}$ can be refined (invariant: $\mathcal{F} = \mathcal{S}$)
  return $(null, \mathcal{F})$

Algorithm 3 Iterative **Refine**

input:

- $\mathcal{Q}$: the package query to be evaluated
- $\mathcal{P} = \{(G_1, \tilde{t}_1), \ldots, (G_m, \tilde{t}_m)\}$: partitioning groups
- $\mathcal{S}$: partitioning groups yet to be refined (initially $\mathcal{S} = \mathcal{P}$)
- $p_{\mathcal{S}}$: the refining package (initially the result of **Sketch**)

output: a feasible package containing only tuples, or failure

procedure **Refine**($\mathcal{Q}$, $\mathcal{P}$, $p_{\mathcal{S}}$)
  ▷ Phase 1: Iterative refinements
  while $\mathcal{S} \neq \emptyset$ **do**
    ▷ Solve all unsolved groups $\mathcal{S}$ independently
    $\mathcal{S}' = \mathcal{S}$
    for all $(G_i, \tilde{t}_i) \in \mathcal{S}$ **do**
      ▷ Skip groups that have no representative in $p_{\mathcal{S}}$
      if $\tilde{t}_i \in p_{\mathcal{S}}$ **then**
        $p_i \leftarrow$ **Direct**($\mathcal{Q}_i(p_{\mathcal{S}})$)
        if $\mathcal{Q}_i(p_{\mathcal{S}})$ **is feasible then**
          $\mathcal{S}' = \mathcal{S}' \setminus \{(G_i, \tilde{t}_i)\}$
    if $\mathcal{S}' \subset \mathcal{S}$ **then**
      ▷ Combine independent solutions into $p_{\mathcal{S}}$
      for all $(G_i, \tilde{t}_i) \in \mathcal{S}$ **do**
        $p_{\mathcal{S}} \leftarrow p_{\mathcal{S}} \setminus \{\tilde{t}_i\} \cup p_i$
      $\mathcal{S} \leftarrow \mathcal{S}'$
    else if $\mathcal{S}' = \mathcal{S}$ **then**
      ▷ No progress could be made
      return failure
  ▷ Phase 2: Feasibility adjustment
  if $p_{\mathcal{S}}$ **is infeasible for** $\mathcal{Q}$ **then**
    ▷ Attempt re-refining groups having at least one tuple
    for all $(G_i, \tilde{t}_i) \in \mathcal{P}$ s.t. $p_{\mathcal{S}} \cap G_i \neq \emptyset$ **do**
      $p_i \leftarrow$ **Direct**($\mathcal{Q}_i(p_{\mathcal{S}})$, $p_{\mathcal{S}}$)
      if $\mathcal{Q}_i(p_{\mathcal{S}})$ **is feasible then**
        $p \leftarrow p_{\mathcal{S}} \setminus \{\tilde{t}_i\} \cup p_i$ ▷ Invariant: p is feasible for $\mathcal{Q}$
        return p
    return failure
  return $p_{\mathcal{S}}$

CHAPTER 4

STOCHASTIC PACKAGE QUERY EVALUATION

In most real-world decision-making problems, the data is uncertain. In this chapter, we provide methods for processing *stochastic package queries* (SPQs), in order to solve optimization problems over uncertain data, right where the data resides. Prior work in stochastic programming uses Monte Carlo methods where the original stochastic optimization problem is approximated by a large deterministic optimization problem that incorporates many *scenarios*, i.e., sample realizations of the uncertain data values. For large database tables, however, a huge number of scenarios is required, leading to poor performance and, often, failure of the solver software. This chapter therefore provides a novel **SummarySearch** algorithm that, instead of trying to solve a large deterministic problem, seamlessly approximates it via a sequence of smaller problems defined over carefully crafted *summaries* of the scenarios that accelerate convergence to a feasible and near-optimal solution. Experimental results on our prototype system show that **SummarySearch** can be orders of magnitude faster than prior methods at finding feasible and high-quality packages.

4.1 Introduction

Constrained optimization is central to decision making over a broad range of domains, including finance [82, 75], transportation [41], healthcare [64], the travel industry [46],

Input Table

Stock_Investments (Table)

id	stock	price	sell_in	Gain
1	AAPL	234	1 day	?
2	AAPL	234	1 week	?
3	MSFT	140	1 day	?
4	MSFT	140	1 week	?
5	TSLA	258	1 day	?
6	TSLA	258	1 week	?

Stochastic Package Query (sPaQL)

```
SELECT PACKAGE(*) AS Portfolio
FROM Stock_Investments
SUCH THAT
  SUM(price) ≤ 1000 AND
  SUM(Gain) ≥ −10 WITH PROBABILITY ≥ 0.95
MAXIMIZE EXPECTED SUM(Gain)
```

Output Package

Portfolio (Package)

id	stock	price	sell_in	Gain
3	MSFT	140	1 day	?
3	MSFT	140	1 day	?
6	TSLA	258	1 week	?

"Buy 2 MSFT shares, sell them tomorrow. Buy 1 TSLA share, sell it in 1 week".

Figure 4.1: Example input table for the FINANCIAL PORTFOLIO (top), its stochastic package query expression in sPaQL (bottom left), and an example output package (bottom right) with a description of its meaning for the investor. Stochastic attributes (Gain, in this example) are denoted in small caps and their values are unknown (shown by a question mark). Sample realizations of the uncertain ? values are generated by calls to VG functions.

robotics [54], and engineering [18]. Consider, for example, the following very common investment problem.

Example 8 (Financial Portfolio)**.** Given uncertain predictions for future stock prices based on financial models derived from historical data, an investor wants to invest $1,000 in a set of trades (decisions on which stocks to buy and when to sell them) that will maximize the *expected future gain*, while ensuring that the *loss* (if any) will be lower than $10 with probability at least 95%.

Suppose each row in a table contains a possible stock trade an investor can make: whether to buy one share of a certain stock, and when to sell it back, as shown in the left-hand side of Figure 4.1. The investor wants a "package" of trades—a subset of the input table, with possible repetitions (i.e., multiple shares)—that is *feasible*, in that it satisfies the given constraints (total price at most $1,000 and loss lower than $10 with probability at least 95%), and *optimal*, in that it maximizes an objective (expected future gain). Although the current price of a stock is known—i.e., **price** is a *deterministic* attribute—its future price, and thus the gain obtained after reselling the stock, is *unknown*. In the input table, Gain is a *stochastic attribute*. If the future gains were known, Example 8 would be a deterministic package query (see Section 2.2.2 in Chapter 2), directly solvable as an *Integer Linear Program* (ILP) using off-the-shelf linear solvers such as IBM CPLEX [76], and declaratively expressible in PaQL. Because Gain is stochastic, the investor is solving a *stochastic ILP* instead. In this chapter, we introduce *stochastic package queries* (SPQs), a generalization of package queries that allows uncertainty in the data, thereby allowing specification and solution of stochastic ILP problems.

In Chapter 2 (Section 2.2.2) we introduced sPaQL, a simple language extension to PaQL that allows easy specification of package queries with stochastic constraints and

objectives. We show the sPaQL query for Example 8 in Figure 4.1. The result of the query, on the right-hand side of the figure, is a package that informs the investor about how many trades to buy for each individual stock, and when to plan reselling them to the stock market.

In this chapter, we first describe a **Direct** evaluation strategy of sPaQL queries (Section 4.3) that, similarly to the deterministic case (Section 3.1), directly translates a sPaQL query into an equivalent integer program and employs an off-the-shelf solver to obtain a package solution. This algorithm is straightforward and produces optimal solutions (up to the solver's abilities), but is only applicable under very stringent conditions, one of which being that all the random variables involved in any of the probabilistic constraints (or in the probabilistic objective) are normally distributed, with known (or analytically derivable) means and variances. This condition is already too strict for the Financial Portfolio example, as we discuss below.

Probabilistic databases [43, 149] enable the representation of random variables in a database. The Financial Portfolio, like many other real-world applications, typically uses complex distributions to model uncertainty. For instance, future stock prices are sometimes forecast using lognormal variates based on "geometric Brownian motion" [130] using historical stock price data; alternatively, forecasts can incorporate complex stochastic predictive simulation or machine learning models. For this reason, we base SPQs on the Monte Carlo probabilistic data model (see Section 2.1.2), which offers support for arbitrary distributions via user-defined *variable generation* (VG) functions. To generate a sample realization of the random variables in a database, the system calls the appropriate VG functions. Whereas existing probabilistic databases excel at supporting SQL-like queries under uncertainty, they do not support package-level optimization, and therefore cannot answer SPQs.

The state of the art in solving stochastic ILPs (SILPs) has been developed outside of the database setting, in the field of *stochastic programming* [7, 35, 47]. These techniques approximate the given SILP by a large deterministic ILP (DILP) that simultaneously incorporates multiple *scenarios*. In a Monte Carlo database, a scenario is obtained by generating a realization of every random variable in the table, via a call to each associated VG function; this procedure may be repeated multiple times, generating a set of scenarios that are mutually independent and identically distributed (i.i.d.). Figure 4.2 shows an example of three possible scenarios for the input investment table for Example 8. Roughly speaking, expectations in the SILP are approximated by averages over the scenarios and probabilities by relative frequencies to form the DILP, which is then fed to a standard solver (e.g., CPLEX). The obtained solution approximates the true optimal solution for the SILP; the more scenarios, the better the approximation.

The solution of the DILP, however, may not be feasible with respect to the original SILP, especially if the approximation is based on only a small number of scenarios that do not well represent the true uncertainty distribution. For example, a financial package obtained by using too few scenarios might guarantee a loss less than \$10 with a probability of only 65%, rather then 95%, incurring more risk than desired.

There is no practical way to know how many scenarios will be needed *a priori*; existing theoretical a-priori bounds—see, e.g., [99]—are usually too conservative to be usable when table sizes are large. For example, if the Stock_Investments table contains N=50,000 rows, then to guarantee that the DILP solution is feasible for the SILP with merely 0.1% probability (which is really no guarantee at all), the theory would indicate that 690,000 scenarios are needed, resulting in a DILP with 34.5 *billion* coefficients! Stochastic programming solutions must therefore be "validated" *a posteriori*, using a much larger, and out-of-sample, set of scenarios. In Example 8, for instance, we would generate, say, 10^6

Scenario 1

id	...	gain
1	...	0.1
2	...	0.05
3	...	-0.2
4	...	0.2
5	...	0.1
6	...	-0.7

Scenario 2

id	...	gain
1	...	-0.2
2	...	-0.03
3	...	0.5
4	...	0.7
5	...	-0.7
6	...	-0.001

Scenario 3

id	...	gain
1	...	0.01
2	...	0.02
3	...	-0.1
4	...	-0.3
5	...	0.2
6	...	0.3

Figure 4.2: Three example scenarios for the Stock_Investments table, each showing only the ids and specific realizations for the stochastic attribute Gain.

scenarios and verify that the loss is less than \$10 in at least 95% of them; such validation is much faster than solving a DILP with 10^6 scenarios.

The state-of-the-art algorithm thus works in a loop: the *optimization phase* creates scenarios, combines them into a DILP, and computes a solution; the *validation phase* validates the solution against the out-of-sample scenarios. If the solution is feasible on the validation scenarios (*validation-feasible*), the algorithm terminates, otherwise it creates more scenarios and repeats. A solution that is validation-feasible is highly likely to be truly feasible for the original SILP. Typically the ultimate number of scenarios used to compute the optimal solution to the DILP is astronomically smaller than the number prescribed by the conservative theoretical bounds (though it is still large enough to be extremely computationally challenging).

Unfortunately, this process often breaks down in practice. Uncertainty increases with increasing table size, and large tables typically need a huge number of scenarios to achieve feasibility. Thus the validation phase repeatedly fails, and the scenario set—and hence the DILP—grows larger and larger until the solver is overwhelmed. Even if the solver can ultimately handle the problem, many ever-slower iterations may be required until validation-feasible solutions are found, resulting in poor performance.

In this chapter, we present an end-to-end system for SPQs, seamlessly connecting SILP optimization with data management and stochastic predictive modeling. Like in the deterministic case, tasks related to efficiently storing data, maintaining consistency, controlling access, and efficiently retrieving and preparing the data for analysis can leverage the full power of a DBMS, while avoiding the usual slow, cumbersome, and error-prone analytics workflow where we read a dataset off of a database into main memory, feed it to stochastic-prediction and optimization packages, and store the results back into the database.

We first introduce a **Naïve** query evaluation algorithm, which embodies the state-of-the-art optimization/validation technique outlined above, and thoroughly discuss its drawbacks. (Although the **Naïve** technique is mentioned in the stochastic programming literature, to our knowledge this is the first systematic implementation of the approach.) We then introduce our new algorithm, **SummarySearch**, that is typically faster than **Naïve** by orders of magnitude and can handle problems that cause **Naïve** to fail.

Our key observation is that the randomly selected set of scenarios used to form the DILP during an iteration of **Naïve** tend to be overly "optimistic", leading the solver towards a seemingly good solution that "in reality"—i.e., when tested against the validation scenarios—turns out to be infeasible. This problem is also known as the "optimizer's curse" [145].

To overcome the optimizer's curse, **SummarySearch** replaces the large set of scenarios used to form the **Naïve** DILP by a very small synopsis of the scenario set, called a "summary", which results in a "reduced" DILP that is much smaller than the **Naïve** DILP. A summary is carefully crafted to be "conservative" in that the constraints in the reduced DILP are harder to satisfy than the constraints in the **Naïve** DILP. Because the reduced DILP is much smaller than the **Naïve** DILP, it can be solved much faster; moreover, the

resulting solution is much more likely to be validation-feasible, so that the required number of optimization/validation iterations is typically reduced. Of course, if a summary is overly conservative, the resulting solution will be feasible, but highly suboptimal. Therefore, during each optimization phase, **SummarySearch** implements a sophisticated search procedure aimed at finding a "minimally" conservative summary; this search requires solution of a sequence of reduced DILPs, but each can be solved quickly.

Our experiments (Section 5.2.2) show that, since its iterations are much faster than those of **Naïve**, **SummarySearch** exhibits a large net performance gain even when the number of iterations is comparable; typically, the number of iterations is actually much lower for **SummarySearch** than for **Naïve**, further augmenting the performance gain.

In summary, in this chapter: We provide a precise and concrete embodiment, the **Naïve** algorithm, of the optimization/validation procedures first introduced in the stochastic programming literature (Section 4.4); We provide a novel algorithm, **SummarySearch**, that is orders-of-magnitude faster than **Naïve**, and that can solve SPQs that require too many scenarios for **Naïve** to handle. This is a significant contribution and fundamental extension to the known state-of-the-art in stochastic programming (Section 4.5); We present techniques that allow **SummarySearch** to optimize its parameters automatically, and we provide theoretical approximation guarantees on the solution of **SummarySearch** relative to **Naïve** (Section 4.6); We provide a comprehensive experimental study, which indicates that **SummarySearch** always finds validation-feasible solutions of high quality, even when **Naïve** cannot, with dramatic speed-ups relative to **Naïve** (Section 5.2.2); We conclude in Section 4.9. Our SPQ techniques represent a significant step towards data-intensive decision making under uncertainty.

4.2 Preliminaries

The work presented in this chapter lies at the intersection of package queries, probabilistic databases, and stochastic programming. In this section, we introduce some basic definitions from these areas that we will use throughout the chapter. For convenience, we slightly simplify some notation previously used for deterministic package queries.

4.2.1 Deterministic package queries

A *package* P of a relation R is a relation obtained from R by inserting $m_{\mathsf{P}}(t) \geq 0$ copies of t into P for each $t \in \mathsf{R}$; here m_{P} is the *multiplicity function* of P. The goal of a *package query* is to specify m_{P}, and hence the tuples of the corresponding package relation. A package query may include a WHERE clause (tuple-level constraints), a SUCH THAT clause (package-level constraints), a package-level objective predicate and, possibly, a REPEAT limit, i.e., an upper bound on the number of duplicates of each tuple in the package.

A deterministic package query can be translated into an equivalent *integer program* [23]. For each tuple $t_i \in \mathsf{R}$, the translation assigns a nonnegative integer decision variable x_i corresponding to the multiplicity of t_i in P, i.e., $x_i = m_{\mathsf{P}}(t_i)$. If the objective function and all constraints are linear in the x_i's, the resulting integer program is an ILP. A cardinality constraint $\mathsf{COUNT}(*) = 3$ is translated into the ILP constraint $\sum_{i=1}^{N} x_i = 3$. A summation constraint $\mathsf{SUM}(\mathbf{price}) \leq 1000$ is translated into $\sum_{i=1}^{N} t_i.\mathbf{price}\, x_i \leq 1000$; this translation works similarly for other linear constraints and objectives. A REPEAT l constraint is translated into bound constraints $x_i \leq l+1, \forall i \in [1..N]$.

4.2.2 Monte Carlo relations

Recall (Section 2.1.2) that we use the Monte Carlo database model [149] to represent uncertainty in a probabilistic database. Uncertain values are modeled as random variables,

and a scenario (a deterministic realization of the relation) is generated by invoking all of the associated VG functions for the relation. In the simplest case, where all random variables are statistically independent, each random variable has its own VG function; in general, multiple random variables can share the same VG function, allowing specification of various kinds of statistical correlations. A Monte Carlo database system such as MCDB [80, 79] (or its successor, SimSQL [29]) facilitates specification of VG functions as user-defined functions. We assume that there exists a deterministic key column that is the same in each scenario, so that each scenario contains exactly N tuples for some $N \geq 1$ and the notion of the "ith tuple t_i" is well defined across scenarios. For simplicity, we focus henceforth on the case where a database comprises a single relation. Our results extend to Monte Carlo databases containing multiple (stochastic) base relations in which the SPQ is defined in terms of a relation obtained via a query over the base relations.

4.2.3 Stochastic ILPs

The field of stochastic programming [139, 85] studies optimization problems—selecting values of decision variables, subject to constraints, to optimize an objective value—having uncertainty in the data. We focus on SILPs with linear constraints and linear objectives that are deterministic, expressed as expectations, or expressed as probabilities. Probabilistic constraints are also called "chance" constraints in the stochastic programming literature.

Linear constraints. Given random variables $\xi_1, \ldots, \xi_N$, decision variables $x_1, \ldots x_N$, a real number $v \in \mathbb{R}$, and a relation $\odot \in \{\leq, \geq\}$, a linear *expectation constraint* takes the form $\mathbb{E}\left(\sum_{i=1}^{N} \xi_i x_i\right) \odot v$, and a linear *probabilistic constraint* takes the form $\Pr\left(\sum_{i=1}^{N} \xi_i x_i \odot v\right) \geq p$, where $p \in [0,1]$. We refer to $\sum_{i=1}^{N} \xi_i x_i \odot v$ as the *inner constraint* of the probabilistic constraint, and to $\sum_{i=1}^{N} \xi_i x_i$ as its *inner function.* As discussed in Section 2.2.2, constraints of the form $\Pr(\cdot) \leq p$ can be rewritten in the aforementioned form by flipping the inequality

sign of the inner constraint and using $1-p$ instead. If for constants $c_1, \ldots, c_N \in \mathbb{R}$ we have $\Pr(\xi_i = c_i) = 1$ for $i \in [1..N]$, then we obtain the deterministic constraint $\sum_{i=1}^{N} c_i x_i \odot v$ as a special case of an expectation constraint.

Objective. Without loss of generality, we assume throughout that the objective has the canonical form $\min_x \sum_{i=1}^{N} c_i x_i$ for deterministic constants $c_1, \ldots, c_N$. Indeed, observe that an objective in the form of an expectation of a linear function can be written in canonical form: $\min_x \mathbb{E}\left(\sum_{i=1}^{N} \xi_i x_i\right) = \min_x \sum_{i=1}^{N} \mathbb{E}(\xi_i) x_i$, and thus we take $c_i = \mathbb{E}(\xi_i)$. (This assumes that each expectation $\mathbb{E}(\xi_i)$ is known or can be accurately approximated.) We call $\sum_{i=1}^{N} \xi_i x_i$ the *inner function* of the expectation. Similarly, an objective in the form of a probability can be written in canonical form using epigraphic rewriting [34]. For example, we can rewrite an objective of the form $\min_x \Pr\left(\sum_{i=1}^{N} \xi_i x_i \odot v\right)$ in canonical form as $\min_{x,y} y$ and add a new probabilistic constraint $\Pr\left(\sum_{i=1}^{N} \xi_i x_i \odot v\right) \leq y$. Here $c_1 = \cdots c_N = 0$ and y is an artificial decision variable added to the problem with objective coefficient $c_y = 1$. Throughout the rest of the chapter, we will primarily focus on techniques for minimization problems with a nonnegative objective function; the various other possibilities can be handled with suitable modifications and are presented in Section 4.6.4.

In our database setting, we assume for ease of exposition that, in a given constraint or objective, each random variable ξ_i corresponds to a random attribute value $t_i.\mathsf{A}$ for some real-valued attribute A; a different attribute can be used for each constraint, and need not be the same as the attribute that appears in the objective. Our methods can actually support more general formulations: e.g., an expectation objective of the form $\min_x \mathbb{E}\left(\sum_{i=1}^{N} g(t_i) x_i\right)$, where g is an arbitrary real-valued function of tuple attributes; constraints can similarly be generalized. Note that this general form allows categorical attributes to be used in addition to real-valued attributes.

Algorithm 4 Naïve Monte Carlo Query Evaluation

$\mathcal{Q}$: A stochastic package query
$\hat{M}$: Number of out-of-sample validation scenarios (e.g., 10^6)
M : Initial number of optimization scenarios (e.g., 100)
m : Iterative increment to M (e.g., 100)
output: A feasible package solution x, or failure (no solution).

1: $\mathcal{S} \leftarrow$ GenerateScenarios$(\mathcal{Q}, M)$ ▷ *Optimization scenarios*
2: **repeat**
3: $\text{SAA}_{\mathcal{Q},M} \leftarrow$ FormulateSAA$(\mathcal{Q}, \mathcal{S})$ ▷ *Approximate DILP*
4: $x \leftarrow$ Solve$(\text{SAA}_{\mathcal{Q},M})$ ▷ *Solve* SAA *with* M *scenarios*
5: $\hat{v}_x \leftarrow$ Validate$(x, \mathcal{Q}, \hat{M})$ ▷ *Validate* x *using* $\hat{M}$ *scenarios*
6: **if** $\hat{v}_x$.is_feasible **then** ▷ x *is feasible*
7: **return** x
8: ▷ *Otherwise, use more optimization scenarios*
9: $\mathcal{S} \leftarrow \mathcal{S} \cup$ GenerateScenarios$(\mathcal{Q}, m)$
10: $M \leftarrow M + m$
11: **until**

4.3 Direct formulation of sPaQL queries

The **Direct** formulation for sPaQL queries follows the same steps as the **Direct** formulation of their deterministic counterpart (Section 3.1), additionally using the translation rules from Section 2.4.2 for expectation and probabilistic predicates. We remind the reader that this formulation is only possible under very stringent conditions (Section 2.4.2).

We now focus on the general cases of non-Gaussian, complex, or unknown distributions, for which no *exact* translation exists. For these, we only have *approximate* formulations and methods.

4.4 Naïve SILP approximation

Recall that **Naïve** is the first systematic implementation of the optimization/validation approach mentioned in the stochastic programming literature. The pseudocode is given as Algorithm 4. As discussed previously, the algorithm generates scenarios (line 1), combines them into an approximating DILP (line 3), solves the DILP to obtain a solution x (line 4),

and then validates the feasibility of x against a large number of out-of-sample validation scenarios (line 5). The process is iterated, adding additional scenarios at each iteration (line 10) until the validation phase succeeds. We now describe these steps in more detail.

As discussed in the Introduction, the optimization phase for the DILP can be very slow, and often the convergence to feasibility requires so many optimize/validate iterations that the DILP becomes too large for the solver to handle, so that **Naïve** fails. Our novel **SummarySearch** algorithm in Section 4.5 uses "summaries" to speed up the optimization phase and reduce the number of required iterations.

4.4.1 Sample-average approximation

As mentioned previously, we can generate a scenario by invoking all of the VG functions for a table to obtain a realization of each random variable, and can repeat this process M times to obtain a Monte Carlo sample of M i.i.d. scenarios. In our implementation, **Naïve** generates scenarios by seeding the random number generator once for the entire execution, and accumulates scenarios in main memory.

We then obtain the DILP from the original SILP by replacing the distributions of the random variables with the empirical distributions corresponding to the sample. That is, the probability of an event is approximated by its relative frequency in the sample, and the expectation of a random variable by its sample average. In the stochastic programming literature, this approach is known as *Sample Average Approximation* (SAA) [7, 99], and we therefore refer to the DILP for the stochastic package query $\mathcal{Q}$ as $\text{SAA}_{\mathcal{Q},M}$.

More formally, suppose that we have M scenarios $S_1, \ldots, S_M$, each with N tuples. Recall that $t_i.\mathsf{A}$ denotes the random variable corresponding to attribute A in tuple t_i, and denote by $s_{ij}.\mathsf{A} \in \mathbb{R}$ the realized value of $t_i.\mathsf{A}$ in scenario S_j. Then each expected

sum $\mathbb{E}\left(\sum_{i=1}^{N} t_i.\mathsf{A}\, x_i\right) = \sum_{i=1}^{N} \mathbb{E}\,(t_i.\mathsf{A})\, x_i$ is approximated by $\sum_{i=1}^{N} t_i.\bar{\mu}_{\mathsf{A}}\, x_i$, where $t_i.\bar{\mu}_{\mathsf{A}} = (1/M)\sum_{j=1}^{M} s_{ij}.\mathsf{A}$.

To approximate a probabilistic constraint of the form

$$\Pr\left(\sum_{i=1}^{N} t_i.\mathsf{A}\, x_i \odot v\right) \geq p, \tag{4.1}$$

we add to the problem a new *indicator variable*, $y_j \in \{0,1\}$ for each scenario $j \in [1..M]$, along with an associated *indicator constraint*: $y_j = \mathbb{1}\left(\sum_{i=1}^{N} s_{ij}.\mathsf{A}\, x_i \odot v\right)$, where the indicator function $\mathbb{1}\,(\cdot)$ equals 1 if the inner constraint is satisfied and equals 0 otherwise. We say that solution x "satisfies scenario S_j" (with respect to the constraint) if and only if $y_j = 1$. (Solvers like CPLEX can handle indicator constraints.) Finally, we add the following linear constraint over the indicator variables: $\sum_{j=1}^{M} y_j \geq \lceil pM \rceil$, where $\lceil u \rceil$ is the smallest integer greater than or equal to u. That is, we require that the solution x satisfies at least a fraction p of the M scenarios. The FORMULATESAA() function applies these approximations to create the DILP $\mathrm{SAA}_{\mathcal{Q},M}$.

Size complexity. With K constraints, the size of $\mathrm{SAA}_{\mathcal{Q},M}$, measured with respect to the number of coefficients, is $\Theta(NMK)$: we have N coefficients for each expectation constraint and, for each probabilistic constraint, $N+1$ coefficients (for $x_1, \ldots, x_N, y_j$) for each scenario.

4.4.2 Out-of-sample validation

After using M scenarios to create and solve the DILP $\mathrm{SAA}_{\mathcal{Q},M}$, we check to see if the solution x is *validation-feasible* in that it is a feasible solution for the DILP $\mathrm{SAA}_{\mathcal{Q},\hat{M}}$ that is constructed using $\hat{M} \gg M$ out-of-sample scenarios. When $\hat{M}$ is sufficiently large, validation feasibility is a proxy for true feasibility, i.e., feasibility for the original SILP; commonly, $\hat{M} = 10^6$ or 10^7. This definition of validation-feasibility is simple, but widely accepted [99].

Although there are other, more sophisticated ways to use validation scenarios to obtain confidence intervals on degree of constraint violation—see, e.g., [34]—these are orthogonal to the scope of this work. Henceforth, we use the term "feasibility" to refer to "validation feasibility", unless otherwise noted.

In our implementation, during a precomputation phase, we actually average $\hat{M} \gg M$ scenarios—the same number as the number of validation scenarios—to estimate each $\mathbb{E}(t_i.\mathsf{A})$; we then append these estimates, denoted $t_i.\hat{\mu}_{\mathsf{A}}$, to the table. We do this because such averaging is typically very fast to execute, and is space-efficient in that we simply maintain running averages. Thus a solution x returned by a solver is always feasible for every expectation constraint, and hence is feasible overall if and only if, for every probabilistic constraint of the form (4.1), x satisfies at least a fraction p of the validation scenarios. We can therefore focus attention on the probabilistic constraints, which are the most challenging.

The procedure VALIDATE$(x, \mathcal{Q}, \hat{M})$ checks the feasibility of x, the solution to $\text{SAA}_{\mathcal{Q},M}$; we describe its operation on a single probabilistic constraint $\Pr\left(\sum_{i=1}^{N} t_i.\mathsf{A}\, x_i \odot v\right) \geq p$, but the same steps are taken independently for each probabilistic constraint. It first seeds the system random number generator with a different seed than the one used to generate the optimization scenarios. For each $j \in [1..\hat{M}]$, it generates a realization $\hat{s}_{ij}.\mathsf{A}$ for each $t_i.\mathsf{A}$ such that $x_i > 0$ (i.e., for each tuple that appears in the solution package), and computes the "score" $\sigma_j = \sum_{i:x_i>0} \hat{s}_{ij}.\mathsf{A}\, x_i$. It then sets $y_j = \mathbb{1}(\sigma_j \odot v)$. After all scenarios have been processed, it computes $Y = \sum_{j=1}^{\hat{M}} y_j$ and declares x to be feasible if $Y \geq \lceil p\hat{M} \rceil$. The algorithm purges all realizations from main memory after each scenario has been processed, and only stores the running count of the y_j's, allowing it to scale to an arbitrary number of validation scenarios. Moreover, a package typically contains a realtively small number of tuples, so only a small number of realizations need be generated.

4.5 Summary-based approximation

The **Naïve** algorithm has three major drawbacks. (1) The overall time to derive a feasible solution to $\text{SAA}_{\mathcal{Q},M}$ can be unacceptably long, since the size of $\text{SAA}_{\mathcal{Q},M}$ sharply increases as M increases. (2) It often fails to obtain a feasible solution altogether—in our experiments, the solver (CPLEX) started failing with just a few hundred optimization scenarios. (3) **Naïve** does not offer any guarantees on how close the objective value ω of the solution x to $\text{SAA}_{\mathcal{Q},M}$ is to the true objective value $\hat{\omega}$ of the solution $\hat{x}$ to the DILP $\text{SAA}_{\mathcal{Q},\hat{M}}$ that is based on the validation scenarios. (Recall that we use $\text{SAA}_{\mathcal{Q},\hat{M}}$ as a proxy for the actual SILP.) A feasible solution x that **Naïve** provides can be far from optimal.

Our improved algorithm, **SummarySearch**, which we present in this section, addresses these challenges by ensuring the efficient generation of feasible results through much smaller "reduced" DILPs that each replace a large collection of M scenarios with a very small number Z of scenario "summaries"; in many cases it suffices to take $Z = 1$. We call such a reduced DILP a *Conservative Summary Approximation* (CSA), in contrast to the much larger sample average approximation (SAA) used by **Naïve**. The summaries are carefully designed to be more "conservative" than the original scenario sets that they replace: the constraints are harder to satisfy, and thus the solver is induced to produce feasible solutions faster. **SummarySearch** also guarantees that, for any user-specified approximation error $\epsilon \geq \epsilon_{\min}$ (where $\epsilon_{\min}$ is defined in Section 4.6.4), if the algorithm returns a solution x, then the corresponding objective value ω satisfies $\omega \leq (1+\epsilon)\hat{\omega}$; in this case we say that x is a $(1+\epsilon)$-*approximate* solution. (Recall that we focus on minimization problems with nonnegative objective functions; the other cases are discussed in Section 4.6.4.)

4.5.1 Conservative Summary Approximation

We first define the concept of an α-summary, and then describe how α-summaries are used to construct a CSA.

Summaries. Recall that a solution x to $\text{SAA}_{\mathcal{Q},M}$ *satisfies* a scenario S_j with respect to a probabilistic constraint of the form of Equation (4.1) if $y_j = \mathbb{1}\left(\sum_{i=1}^{N} s_{ij}.\mathsf{A}\, x_i \odot v\right) = 1$, where $s_{ij}.\mathsf{A}$ is the realized value of $t_i.\mathsf{A}$ in S_j.

Definition 10 (α-SUMMARY)**.** *Let $\alpha \in [0,1]$. An* α-summary $S = \{s_i.\mathsf{A} : 1 \le i \le N\}$ *of a scenario set $\mathcal{S} = \{S_1, \ldots, S_M\}$ with respect to a probabilistic constraint C of the form* (4.1) *is a collection of N deterministic values of attribute A such that if a solution x* satisfies *S in that $\sum_{i=1}^{N} s_i.\mathsf{A}\, x_i \odot v$, then x satisfies at least $\lceil \alpha M \rceil$ of the scenarios in $\mathcal{S}$ with respect to C.*

Constructing an α-summary, for $\alpha > 0$, is simple: Suppose that the inner constraint of probabilistic constraint C has the form $\sum_{i=1}^{N} t_i.\mathsf{A}\, x_i \ge v$. Given any subset of scenarios $G(\alpha) \subseteq \mathcal{S}$ of size exactly $\lceil \alpha M \rceil$, we define S as the tuple-wise minimum over $G(\alpha)$:

$$s_i.\mathsf{A} \coloneqq \min_{S_j \in G(\alpha)} s_{ij}.\mathsf{A}$$

Proposition 2. *S is an α-summary of $\mathcal{S}$ with respect to C.*

Proof. Suppose x satisfies S, i.e., $\sum_{i=1}^{N} s_i.\mathsf{A}\, x_i \ge v$. Then, for every scenario $S_j \in G(\alpha)$, $\sum_{i=1}^{N} s_{ij}.\mathsf{A}x_i \ge \sum_{i=1}^{N} s_i.\mathsf{A}x_i \ge v$. Since $|G(\alpha)| = \lceil \alpha M \rceil$, the result follows. ■

Figure 4.3 illustrates an α-summary for the three scenarios in Figure 4.2, where $\alpha = 0.66$ and $G(\alpha)$ comprises scenarios 1 and 3. The summary is conservative in that, for any

Scenario 1

id	...	gain
1	...	0.1
2	...	0.05
3	...	-0.2
4	...	0.2
5	...	0.1
6	...	-0.7

Scenario 3

id	...	gain
1	...	0.01
2	...	0.02
3	...	-0.1
4	...	-0.3
5	...	0.2
6	...	0.3

0.66-Summary

id	...	gain
1	...	0.01
2	...	0.02
3	...	-0.2
4	...	-0.3
5	...	0.1
6	...	-0.7

Figure 4.3: Using two out of the three scenarios of Figure 4.2, we derive a 0.66-summary.

choice x of trades, the gain under the summary values will be less than the gain under either of the two scenarios. Thus if we can find a solution that satisfies the summary, it will automatically satisfy at least scenarios 1 and 3. It might also satisfy scenario 2, and possibly many more scenarios, including unseen scenarios in the validation set. Indeed, if we are lucky, and in fact our solution satisfies at least $100p\%$ of the scenarios in the validation set, then x will be feasible with respect to the constraint on Gain.

Clearly, for an inner constraint with $\leq$, the tuple-wise *maximum* of $G(\alpha)$ yields an α-summary. While there may be other ways to construct α-summaries, in this thesis we only consider minimum and maximum summaries, and defer the study of other, more sophisticated summarization methods to future work. Importantly, a summary need not coincide with any of the scenarios in $\mathcal{S}$; we are exploiting the fact that optimization and validation are decoupled.

CSA formulation. A CSA is basically an SAA in which all probabilistic constraints are approximated using summaries instead of scenarios.[1] The foregoing development implicitly assumed a single summary (with respect to a given probabilistic constraint C) for all of the

[1]As with the SAA formulation, expectations are approximated as averages over a huge number $\hat{M}$ of independent scenarios.

M scenarios in $\mathcal{S}$. In general, we use Z summaries, where $Z \in [1..M]$. These are obtained by dividing $\mathcal{S}$ randomly into Z disjoint partitions $\Pi_1, \ldots, \Pi_Z$, of approximately M/Z scenarios each. Then the α-summary $S_z = \{s_{iz}.\mathsf{A} : 1 \leq i \leq N\}$ for partition Π_z is obtained by taking a tuple-wise minimum or maximum over scenarios in a subset $G_z(\alpha) \subseteq \Pi_z$, where $|G_z(\alpha)| = \lceil \alpha |\Pi_z| \rceil$.

For each probabilistic constraint C of form (4.1), we add to the DILP a new indicator variable, $y_z \in \{0, 1\}$, and an associated indicator constraint $y_z := \mathbb{1}\left(\sum_{i=1}^{N} s_{iz}.\mathsf{A}\, x_i \odot v\right)$. We say that solution x "satisfies summary S_z" iff $y_z = 1$. We also add the linear constraint $\sum_{z=1}^{Z} y_z \geq \lceil pZ \rceil$, requiring at least $100p\%$ of the summaries to be satisfied. We denote the resulting reduced DILP by $\text{CSA}_{\mathcal{Q},M,Z}$.

Size complexity. Assuming K probabilistic constraints, the number of coefficients in $\text{CSA}_{\mathcal{Q},M,Z}$ is $\Theta(NZK)$, which is independent of M. Usually, Z takes on only small values, so that the effective size complexity is only $\Theta(NK)$.

Our results (Section 5.2.2) show that in most cases **SummarySearch** finds good solutions with only one summary, i.e., $Z = 1$. Because Z is small, the solution to $\text{CSA}_{\mathcal{Q},M,Z}$ can be rapidly computed by a solver. The CSA formulation is also more robust to random fluctuations in the sampled data values, and less prone to "overfit" to an unrepresentative set of scenarios obtained by luck of the draw.

An important observation is that as Z increases, $\text{CSA}_{\mathcal{Q},M,Z}$ approaches the $\text{SAA}_{\mathcal{Q},M}$ formulation: at $Z = M$ each partition will contain exactly one scenario, which will also coincide with the summary for the partition. Since $\text{CSA}_{\mathcal{Q},M,Z}$ encompasses $\text{SAA}_{\mathcal{Q},M}$, we can always do at least as well as **Naïve** with respect to the feasibility and optimality properties of our solution, given M scenarios. We address the issue of how to choose Z,

Algorithm 5 SUMMARYSEARCH Query Evaluation

$\mathcal{Q}$: A stochastic package query with K probabilistic constraints
$\mathcal{Q}_0$: $\mathcal{Q}$ devoid of all probabilistic constraints
$\hat{M}$: Number of out-of-sample validation scenarios (e.g., 10^6)
M : Initial number of optimization scenarios (e.g., 100)
m : Iterative increment to M (e.g., 100)
z : Iterative increment to Z (e.g., 1)
ϵ : User-defined approximation error bound, $\epsilon \geq \epsilon_{\min}$
output: A feasible package solution x, or failure (no solution).

▷ *Solve probabilistically-unconstrained problem*
$x^{(0)} \leftarrow \text{SOLVE}(\text{SAA}(\mathcal{Q}_0, \hat{M}))$
$Z = 1$ ▷ Initial number of summaries
repeat
  $(x, \hat{v}_x) \leftarrow \text{CSA-SOLVE}(\mathcal{Q}, x^{(0)}, M, Z)$
  if $\hat{v}_x$.is_feasible **and** $\hat{v}_x$.upper_bound $\leq \epsilon$ **then**
    return x ▷ x is feasible and $(1+\epsilon)$-approximate
  else if $\hat{v}_x$.is_feasible **and** $Z < M$ **then**
    $Z \leftarrow Z + \min\{z, M - Z\}$ ▷ Use more summaries
  else
    $M \leftarrow M + m$ ▷ Use more scenarios
until

α, and each $G_z(\alpha)$ below and in Section 4.6, and also discuss how to generate summaries efficiently.

4.5.2 Query Evaluation with CSA

Algorithm 5 shows query evaluation with **SUMMARYSEARCH**. The goal is to find a feasible solution whose objective value is as close as possible to $\hat{\omega}$, the objective value of the SAA based on the $\hat{M}$ validation scenarios. In the algorithm, $\mathcal{Q}_0$ denotes the SPQ obtained from $\mathcal{Q}$ by removing all of the probabilistic constraints. At the first step, **SUMMARYSEARCH** computes $x^{(0)}$, the solution to the DILP $\text{SAA}_{\mathcal{Q}_0, \hat{M}}$; the only constraints are deterministic constraints and expectation constraints, with the latter estimated from $\hat{M}$ scenarios in the usual way. This corresponds to the "least conservative" solution possible, and is effectively equivalent to solving a CSA using summaries constructed with $\alpha = 0$,

because 0% (i.e., none) of the scenarios are required to be satisfied. For some problems, $x^{(0)}$ might have an infinite objective value, in which case we simply ignore this solution and incrementally increase α until we find a finite solution.

Like **Naïve**, the **SummarySearch** algorithm starts with an initial number of optimization scenarios, $M \geq 1$, and iteratively increments it while solutions are infeasible. In the optimization phase, the algorithm uses a CSA formulation, which replaces the M real scenarios with Z conservative summaries. Initially, the algorithm uses $Z = 1$, replacing the set of M scenarios with a single summary. After feasibility is achieved for a solution x with objective value ω_x, the algorithm tries to check whether the ratio $\epsilon_x = (\omega_x - \hat{\omega})/\hat{\omega}$ is less than or equal to the user-defined error bound ϵ; although $\hat{\omega}$, and hence ϵ_x, is unknown, we can conservatively check whether $\epsilon'_x \leq \epsilon$, where ϵ'_x is an upper bound on ϵ_x that we develop in Section 4.6.4, If the solution is unsatisfactory, **SummarySearch** increases Z, and iterates again. The algorithm stops if and when a feasible and $(1+\epsilon)$-approximate solution is found. In practice, because of the conservative nature of summaries, **SummarySearch** typically finds feasible solutions in drastically fewer iterations than **Naïve**.

4.6 Optimal summary selection

The key component of **SummarySearch** is CSA-Solve, described in this section. With M and Z fixed, CSA-Solve finds the best CSA formulation, i.e., the one having, for each constraint, the optimal value of α and the best set $G_z(\alpha)$ of scenarios for each summary. CSA-Solve thus determines the best solution x achievable with M scenarios and Z summaries, and also computes metadata $\hat{v}_x$ used by **SummarySearch** for checking feasibility and optimality.

4.6.1 CSA-Solve Overview

Algorithm 6 depicts the iterative process of CSA-Solve: at each iteration q it produces a solution $x^{(q)}$ to a problem $\text{CSA}_{\mathcal{Q},M,Z}$ based on an $\alpha_k^{(q)}$-summary for each constraint C_k. Initially, $\alpha_k^{(0)} = 0$ for all k, and thus the solution to $\text{CSA}_{\mathcal{Q},M,Z}$ is simply $x^{(0)}$, which has already been computed by **SummarySearch** prior to calling CSA-Solve. Then CSA-Solve stops in two cases: (1) if it finds a feasible $(1+\epsilon)$-approximate solution; (2) if it enters a cycle, producing the same solution twice with the same α_k values. In case (2), it returns the "best" solution found so far: if one or more feasible solutions have been found, it returns the one with the best objective value, otherwise it returns an infeasible solution, and **SummarySearch** will increase M in its next iteration.

4.6.2 Choosing α

Larger α leads to more conservative α-summaries, as we take the tuple-wise minimum (or maximum) over more and more scenarios. Thus a high value of α increases the chances of finding a feasible solution. On the other hand, if the constraints are more restrictive than necessary, then the solution can have a seriously suboptimal objective value because we are considering fewer candidate solutions, possibly missing the best ones. Thus, CSA-Solve seeks the minimally conservative value of α that will suffice.

How can we measure the true conservativeness of α with respect to a constraint $C \coloneqq \Pr(\sum_{i=1}^{N} t_i.\boldsymbol{A}\, x_i \odot v) \geq p$? As discussed previously, the solution x to a formulation $\text{SAA}_{\mathcal{Q},M}$ based on α-summaries is guaranteed to satisfy at least $100\alpha\%$ of the M optimization scenarios, but the actual true probability of satisfying the constraint—or more pragmatically, the fraction of the $\hat{M}$ validation scenarios satisfied by x—will usually differ from α. Thus,

we look at the difference between the fraction of validation scenarios satisfied by x and the target value p. We call this difference the *p-surplus*, and define it as:

$$r = r(\alpha) \coloneqq \left\{ (1/\hat{M}) \sum_{j=1}^{\hat{M}} \mathbb{1}\left(\sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\, x_i \odot v \right) \right\} - p$$

We expect the function $r(\alpha)$ to be increasing in α with high probability.

Observe that x essentially satisfies the constraint $C' \coloneqq \Pr(\sum_{i=1}^{N} t_i.\mathsf{A}\, x_i \odot v) \geq p + r$. Clearly, if $r < 0$, then x is infeasible for constraint C, whereas if $r > 0$, then x satisfies the inner constraint with a probability that exceeds p, and so is conservative and therefore likely suboptimal. Thus the optimal value α^* satisfies $r(\alpha^*) = 0$. Solutions that achieve zero p-surplus may be impossible to find, and therefore CSA-Solve tries to choose $\alpha = (\alpha_1, \ldots, \alpha_K)$ to minimize the p-surplus for each of the K constraints, while keeping it nonnegative. The search space is finite (hence the possibility of cycles) since $\alpha_k \in \{Z/M, 2Z/M, \ldots, 1\}$ for $k \in [1..K]$.

At each iteration q, CSA-Solve updates $\alpha^{(q-1)}$ to $\alpha^{(q)}$, creates the corresponding CSA problem, and produces a new solution $x^{(q)}$. For simplicity and ease of computation, our initial implementation updates each $\alpha_k^{(q)}$ individually by fitting a smooth curve $R_k^{(q)}(\alpha_k)$ to the historical points $(\alpha_k^{(0)}, r_k^{(0)}), \ldots, (\alpha_k^{(q-1)}, r_k^{(q-1)})$ and then solving the equation $R_k^{(q)}(\alpha_k) = 0$. In our experiments, we observed that (1) fitting an arctangent function provides the most accurate predictions and (2) this artificial decoupling with respect to the constraints yields effective summaries; we plan to investigate other methods for jointly updating $(\alpha_1^{(q-1)}, \ldots, \alpha_K^{(q-1)})$.

Algorithm 6 CSA-SOLVE

$\mathcal{Q}$: A stochastic package query with K probabilistic constraints
$x^{(0)}$: Solution of probabilistically-unconstrained problem
M : Number of optimization scenarios
Z : Number of summaries, $1 \leq Z \leq M$
ϵ : User-defined approximation error bound, $\epsilon \geq \epsilon_{\min}$
output: A feasible and $(1+\epsilon)$-approximate solution, or an infeasible solution

$q \leftarrow 0$ ▷ *Iteration count*
$\mathcal{H} \leftarrow \emptyset$ ▷ *Initialize validation history*
$\alpha^{(q)} = (\alpha_1^{(q)}, \ldots, \alpha_K^{(q)}) \leftarrow (0, \ldots, 0)$ ▷ *Initial conservativeness*
repeat
    ▷ *If entered a cycle, return best solution from history*
    if $(x^{(q)}, \alpha^{(q)}) \in \mathcal{H}$ **then**
        return BEST($\{x : (x, \alpha) \in \mathcal{H}\}$)
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{(x^{(q)}, \alpha^{(q)})\}$ ▷ *Update validation history*
    $\hat{v}^{(q)} \leftarrow$ VALIDATE($x^{(q)}, \mathcal{Q}, \hat{M}$) ▷ *Validate & compute metadata*
    $\epsilon^{(q)} \leftarrow \hat{v}^{(q)}$.upper_bound ▷ *Validation upper bound on ϵ*
    for $k = 1, \ldots, K$ **do**
        $r_k^{(q)} \leftarrow \hat{v}_k^{(q)}$.surplus ▷ *Validation p-surplus*
    ▷ *Termination with feasible $(1+\epsilon)$-approximate solution*
    if $\epsilon^{(q)} \leq \epsilon$ **and** $\forall k : r_k^{(q)} \geq 0$ **then**
        return $(x^{(q)}, \hat{v}^{(q)})$
    $q \leftarrow q + 1$ ▷ Iterate again with a new set of summaries
    $\alpha^{(q)} \leftarrow$ GUESSOPTIMALCONSERVATIVENESS($\mathcal{H}$)
    for $k = 1, \ldots, K$ **do**
        $\tilde{S}_k \leftarrow$ SUMMARIZE($x^{(q)}, \alpha_k^{(q)}, C_k, \mathcal{H}$)
    $\text{CSA}_{\mathcal{Q},M,Z} \leftarrow$ FORMULATESAA($\mathcal{Q}, \{\tilde{S}_1, \ldots, \tilde{S}_K\}$)
    $x^{(q)} \leftarrow$ SOLVE($\text{CSA}_{\mathcal{Q},M,Z}$)
until

4.6.3 Choosing $\mathbf{G_z}$

So far, we have assumed that the subset $G_z(\alpha_k^{(q)})$ used to build the summary is *any* set containing $n_k^{(q)} = \lceil \alpha_k^{(q)} |\Pi_z| \rceil$ scenarios. **SUMMARYSEARCH** employs a simple greedy heuristic to determine $G_z(\alpha_k^{(q)})$: it chooses the $n_k^{(q)}$ scenarios that produce the summary most likely to keep the previous solution feasible in the current iteration, so that the new solution will likely have a higher objective value. For an inner $\geq$ ($\leq$) constraint, this is

achieved by sorting the scenarios in Π_z according to their "scenario score" $\sum_{i=1}^{N} s_{ij}.\mathsf{A}\, x_i^{(q-1)}$ and taking the first $n_k^{(q)}$ in descending (ascending) order.

4.6.4 Approximation Guarantees

If $x^{(q)}$ is feasible, **SummarySearch** can terminate if it can determine that $x^{(q)}$ is $(1+\epsilon)$-approximate relative to the optimal feasible solution $\hat{x}$ based on the validation scenarios, i.e., that $\omega^{(q)} \leq (1+\epsilon)\hat{\omega}$, where $\omega^{(q)}$ and $\hat{\omega}$ are the objective values for $x^{(q)}$ and $\hat{x}$, respectively, and ϵ is an accuracy parameter specified by the user. Without loss of generality, we assume below that the objective function is an expectation; should the objective be deterministic, nesting it within an expectation does not change its value.

This termination check proceeds as follows. During the qth iteration of **SummarySearch**, the function Validate$(x^{(q)}, \mathcal{Q}, \hat{M})$ computes p-surplus values $r_1^{(q)}, \ldots, r_K^{(q)}$, one for each probabilistic constraint in the query. Further, it computes $\epsilon^{(q)}$ (as defined below). We show below that if $\epsilon^{(q)} \leq \epsilon$ and $\forall k : r_k^{(q)} \geq 0$, then $x^{(q)}$ is a feasible $(1+\epsilon)$-approximate solution, and **SummarySearch** can immediately return $x^{(q)}$ and terminate. As usual, we focus on minimization problems with nonnegative objective values, and take the optimal solution $\hat{x}$ and objective value $\hat{\omega}$ of $\text{SAA}_{\mathcal{Q},\hat{M}}$ as proxies for those of the original SILP. We start with the following simple but important result.

Proposition 3 (General Approximation Guarantee)**.** *Let $\epsilon \geq 0$ and let $\underline{\omega}$ be a positive constant such that $\underline{\omega} \leq \hat{\omega}$. Set $\epsilon^{(q)} = (\omega^{(q)}/\underline{\omega}) - 1$. If $\epsilon^{(q)} \leq \epsilon$, then $\omega^{(q)} \leq (1+\epsilon)\hat{\omega}$.*

Proof. Suppose that $\epsilon^{(q)} \leq \epsilon$. Since $\hat{\omega}/\underline{\omega} \geq 1$, we have

$$\omega^{(q)} \leq \left(\frac{\hat{\omega}}{\underline{\omega}}\right)\omega^{(q)} = \left(1 + \left(\frac{\omega^{(q)}}{\underline{\omega}} - 1\right)\right)\hat{\omega} = \left(1+\epsilon^{(q)}\right)\hat{\omega} \leq (1+\epsilon)\hat{\omega},$$

and the result follows. ■

We obtain a specific formula for $\epsilon^{(q)}$ by choosing a specific bound $\underline{\omega}$. Clearly, we would like to choose $\underline{\omega}$ as large as possible, since this maximizes the likelihood that $\epsilon^{(q)} \leq \epsilon$. One simple choice that always works is to set $\underline{\omega} = \omega^{(0)}$, where $\omega^{(0)}$ is the objective value of the SAA problem corresponding to the original SILP but with all probabilistic constraints removed—see line 2 of Algorithm 5. If all random variables are lower-bounded by a constant $\underline{s} > 0$ and the size of any feasible package is lower-bounded by a constant $\underline{l} > 0$, then $\sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\, x_i \geq \underline{sl}$, $\forall j \in [1..\hat{M}]$, so that

$$\hat{\omega} = \frac{1}{\hat{M}} \sum_{j=1}^{\hat{M}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\, \hat{x}_i \geq \frac{1}{\hat{M}} \sum_{j=1}^{\hat{M}} \underline{sl} = \underline{sl},$$

which yields an alternative lower bound. Yet another bound can be sometimes obtained by exploiting the relation of the constraints to the objective.

Definition 11 (Objective-Constraint Interaction)**.** *Let the objective be* $\min \mathbb{E}(\sum_{i=1}^{N} \xi_i x_i)$*, for random variables* $\{\xi_i\}_{i \in [1..N]}$*. The objective is said to be* supported *by a constraint of the form* $\Pr\left(\sum_{i=1}^{N} \xi_i x_i \leq v\right) \geq p$ *and* counteracted *by a constraint of the form* $\Pr(\sum_{i=1}^{N} \xi_i x_i \geq v) \geq p$*. All other forms of constraint are said to be* independent *of the objective.*

Intuitively, a supporting probabilistic constraint "supports" the objective function in the same "direction" of the optimization ($\leq$ for minimization, $\geq$ for maximization), whereas a counteracting constraint goes against the optimization. If there exists a counteracting constraint with $v \geq 0$, it can be shown (Section 4.6.4.2) that $\hat{\omega} \geq pv$.

Finally, we take $\underline{\omega}$ to be the maximum of all applicable lower bounds. Similar formulas can be derived for other possible cases—maximization problems, negative objective values, and so on; see Section 4.6.4.1 and Section 4.6.4.2.

Note that if $(\hat{\omega}/\underline{\omega})-1>\epsilon$, then $\epsilon^{(q)}=(\omega^{(q)}/\underline{\omega})-1>\epsilon$, $\forall q\geq 0$, so that **SummarySearch** cannot terminate with a feasible $(1+\epsilon)$-approximate solution. To avoid this problem, we require that $\epsilon\geq\epsilon_{\min}$, where $\epsilon_{\min}=(\overline{\omega}/\underline{\omega})-1$. Here $\overline{\omega}$ is any upper bound on $\hat{\omega}$. It can be shown, for example, that if (1) all random variables are upper-bounded by a constant $\overline{s}>0$, (2) the size of any feasible package is upper-bounded by a constant $\overline{l}>0$, and (3) there exists a supporting constraint with $v\geq 0$, then $\hat{\omega}\leq v+(1-p)\overline{s}\overline{l}$; see Section 4.6.4.2. If we have available a feasible solution x with objective value ω_x, then we can take $\overline{\omega}=\omega_x$. We choose $\overline{\omega}$ to be the minimum of all applicable bounds.

4.6.4.1 Objective types and signs

Recall that in Proposition 3 we assumed a minimization query with nonnegative objective values. The following propositions replace Proposition 3 under different conditions.

Minimization with negative objective values.

Proposition 4. *Let $\epsilon\geq 0$ and let $\underline{\omega}$ be a negative constant such that $\underline{\omega}\leq\hat{\omega}$. Set $\epsilon^{(q)}=(\underline{\omega}/\omega^{(q)})-1$. If $\epsilon^{(q)}\leq\epsilon$, then $\hat{\omega}\geq(1+\epsilon)\omega^{(q)}$.*

Proof. Suppose that $\epsilon^{(q)}\leq\epsilon$. We have

$$\hat{\omega}\geq\underline{\omega}=\left(1+\left(\frac{\underline{\omega}}{\omega^{(q)}}-1\right)\right)\omega^{(q)}=\left(1+\epsilon^{(q)}\right)\omega^{(q)}\geq(1+\epsilon)\omega^{(q)},$$

and the result follows. ■

Maximization with nonnegative objective values.

Proposition 5. *Let $\epsilon\geq 0$ and let $\overline{\omega}$ be a positive constant such that $\hat{\omega}\leq\overline{\omega}$. Set $\epsilon^{(q)}=(\overline{\omega}/\omega^{(q)})-1$. If $\epsilon^{(q)}\leq\epsilon$, then $\hat{\omega}\leq(1+\epsilon)\omega^{(q)}$.*

Proof. Suppose that $\epsilon^{(q)} \leq \epsilon$. We have

$$\hat{\omega} \leq \overline{\omega} = \left(1 + \left(\frac{\overline{\omega}}{\omega^{(q)}} - 1\right)\right)\omega^{(q)} = \left(1 + \epsilon^{(q)}\right)\omega^{(q)} \leq (1+\epsilon)\omega^{(q)},$$

and the result follows. ■

Maximization with negative objective values.

Proposition 6. *Let $\epsilon \geq 0$ and let $\overline{\omega}$ be a negative constant such that $\hat{\omega} \leq \overline{\omega}$. Set $\epsilon^{(q)} = (\omega^{(q)}/\overline{\omega}) - 1$. If $\epsilon^{(q)} \leq \epsilon$, then $\omega^{(q)} \geq (1+\epsilon)\hat{\omega}$.*

Proof. Suppose that $\epsilon^{(q)} \leq \epsilon$. Since $\hat{\omega}/\overline{\omega} \geq 1$ and $\omega^{(q)} < 0$, we have

$$\omega^{(q)} \geq \left(\frac{\hat{\omega}}{\overline{\omega}}\right)\omega^{(q)} = \left(1 + \left(\frac{\omega^{(q)}}{\overline{\omega}} - 1\right)\right)\hat{\omega} = \left(1 + \epsilon^{(q)}\right)\hat{\omega} \geq (1+\epsilon)\hat{\omega},$$

and the result follows. ■

4.6.4.2 Upper and lower bounds on $\hat{\omega}$

Our theory uses upper and lower bounds on the optimal objective value $\hat{\omega}$ to derive approximation bounds. We provide bounds under the following assumptions: (A1) there exist bounds on the values of the validation scenarios, (A2) there exist package size bounds. These two assumptions are not too restrictive since we can almost always find such bounds by analyzing the query, or the validation scenarios produced by the VG functions. The following are examples of simple bounds for (A1) and (A2):

(A1) Validation scenarios bounds. We assume the availability of upper and lower bounds on the values of the validation scenarios across the tuples in the optimal package. That is, there should exist $\underline{s}$ and $\overline{s}$ such that $\underline{s} \leq \hat{s}_{ij}.\mathsf{A} \leq \overline{s}$, for all $i \in [1..N] : \hat{x}_i > 0$, and $j \in$

$[1..\hat{M}]$. We can easily derive (possibly loose) bounds, by taking the minimum and maximum scenario values across all input tuples, i.e., by setting $\underline{s} \coloneqq \min\{\hat{s}_{ij}.\mathsf{A} \mid i \in [1..N], j \in [1..\hat{M}]\}$, and $\overline{s} \coloneqq \max\{\hat{s}_{ij}.\mathsf{A} \mid i \in [1..N], j \in [1..\hat{M}]\}$. In principle, tighter bounds might exist. For example, if we could identify tuples that cannot be part of the optimal solution, we could take them out of the min and max in the above formulas. In this work, we do not explore ways to derive better bounds.

(A2) Package size bounds. We also assume there exist upper and lower bounds on the size of the optimal package. That is, there exist $\underline{l}$ and $\bar{l}$ such that $\underline{l} \leq \sum_{i=1}^{N} \hat{x}_i \leq \bar{l}$. An obvious value for $\underline{l}$, always true, is $\underline{l} = 0$. If the package query includes a cardinality constraint (i.e., a constraint on the $\mathsf{COUNT}(*)$), this might be used directly to derive $\underline{l}$, $\bar{l}$, or both. If the query includes deterministic summation constraints (i.e., on $\mathsf{SUM}(\mathsf{A})$, for a deterministic attribute A), we can derive bounds following the derivations presented in [27]. Again, in this work we do not study ways to derive tighter bounds than the obvious ones.

We provide two types of bounds on $\hat{\omega}$: (B1) constraint-agnostic bounds, which are always available regardless of the probabilistic constraints; (B2) constraint-specific bounds, which depend on the probabilistic constraint and on whether the constraint supports or counteracts the objective function, or it is independent of it (see Definition 11). In cases where both (B1) and (B2) are available, the final bound is the best of the two.

(B1) Constraint-agnostic bounds. In Table 4.1, we show bounds on the optimal objective value of the form $\underline{\omega} \leq \hat{\omega} \leq \overline{\omega}$.

Proof. The proof for $\underline{\omega}$ for case $\underline{s} \geq 0$ was provided in Section 4.6.4. Similar derivations can be used for $\underline{\omega}$ under $\underline{s} < 0$ and for $\overline{\omega}$. For example, for $\underline{\omega}$ under $\underline{s} < 0$, we have:

$$\hat{\omega} = \frac{1}{\hat{M}} \sum_{j=1}^{\hat{M}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\, \hat{x}_i \geq \frac{1}{\hat{M}} \sum_{j=1}^{\hat{M}} \underline{s}\bar{l} = \underline{s}\bar{l},$$

using the fact that $\sum_{i=1}^{N} \hat{x}_i \leq \bar{l}$. The other derivations follow a similar reasoning. ■

(B2) Constraint-specific bounds. Another class of bounds exist for an objective function that is supported or counteracted by at least one probabilistic constraint of the form $\Pr\left(\sum_{i=1}^{N} \xi_i x_i \odot v\right) \geq p$. We first set the followings:

$$\hat{S}^{\odot}_{\hat{x}} \coloneqq \{j \in [1..\hat{M}] \mid \textstyle\sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\ \hat{x}_i \odot v\},$$
$$\hat{S}^{\otimes}_{\hat{x}} \coloneqq [1..\hat{M}] \setminus \hat{S}^{\odot}_{\hat{x}},$$
$$\hat{\omega}^{\odot} \coloneqq \frac{1}{\hat{M}} \sum_{j \in \hat{S}^{\odot}_{\hat{x}}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\ \hat{x}_i,$$
$$\hat{\omega}^{\otimes} \coloneqq \frac{1}{\hat{M}} \sum_{j \in \hat{S}^{\otimes}_{\hat{x}}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\ \hat{x}_i.$$

Intuitively, $\hat{S}^{\odot}_{\hat{x}}$ is the set of validation scenarios satisfied by the optimal solution $\hat{x}$, and $\hat{S}^{\otimes}_{\hat{x}}$ is the set of validation scenarios *not satisfied* by $\hat{x}$. Notice that the optimal value is $\hat{\omega} = \hat{\omega}^{\odot} + \hat{\omega}^{\otimes}$.

We provide bounds in the form $\underline{\omega}^{\odot} + \underline{\omega}^{\otimes} \leq \hat{\omega} \leq \overline{\omega}^{\odot} + \overline{\omega}^{\otimes}$. They are implied by the following conditions:

$$\text{(C1) } \underline{\omega}^{\odot} \leq \hat{\omega}^{\odot} \quad \text{(C2) } \underline{\omega}^{\otimes} \leq \hat{\omega}^{\otimes},$$
$$\text{(C3) } \hat{\omega}^{\odot} \leq \overline{\omega}^{\odot} \quad \text{(C4) } \hat{\omega}^{\otimes} \leq \overline{\omega}^{\otimes}.$$

Table 4.2 shows all the available bounds for (C1-4) under different cases. Recall, from Section 4.6.4, that our algorithm uses the best available bounds (i.e., maximum lower bound, and minimum upper bound) among all the available ones, including the ones in Table 4.1.

Case			$\underline{\omega}$	$\overline{\omega}$
$\underline{s} \geq 0$	or	$\overline{s} \geq 0$	$\underline{s}\underline{l}$	$\overline{s}\overline{l}$
$\underline{s} < 0$	or	$\overline{s} < 0$	$\underline{s}\overline{l}$	$\overline{s}\underline{l}$

Table 4.1: Constraint-agnostic bounds on the optimal objective value ($\underline{\omega} \leq \hat{\omega} \leq \overline{\omega}$) that lead to $(1+\epsilon)$-approximations. These bounds are defined over existing bounds on the validation scenarios and package size: $\underline{s} \leq \hat{s}_{ij} \leq \overline{s}$, for all $i \in [1..N]$ and $j \in [1..\hat{M}]$, and $0 \leq \underline{l} \leq \sum_{i=1}^{N} \hat{x}_i \leq \overline{l}$.

Obj.-Constraint Interaction	Case	$\underline{\omega}^{\odot}$	$\underline{\omega}^{\otimes}$	$\overline{\omega}^{\odot}$	$\overline{\omega}^{\otimes}$
(a) Independent	$\underline{s} \geq 0$ or $\overline{s} \geq 0$	$p\underline{s}\underline{l}$	0	$\overline{s}\overline{l}$	$(1-p)\overline{s}\overline{l}$
	$\underline{s} < 0$ or $\overline{s} < 0$	$\underline{s}\overline{l}$	$(1-p)\underline{s}\overline{l}$	$p\overline{s}\underline{l}$	0
(b) Supporting/counteracting	$\sum_{i=1}^{N} \xi_i x_i \geq v \geq 0$	pv	0	—	—
	$\sum_{i=1}^{N} \xi_i x_i \geq v, v < 0$	v	$(1-p)v$	—	—
	$\sum_{i=1}^{N} \xi_i x_i \leq v, v \geq 0$	—	—	v	$(1-p)v$
	$\sum_{i=1}^{N} \xi_i x_i \leq v < 0$	—	—	pv	0

Table 4.2: Constraint-specific bounds for an objective with inner function $\sum_{i=1}^{N} \xi_i x_i$ subject to a probabilistic constraint with right-hand side p. For group (a) in this table, the probabilistic constraint is independent of the objective function; for group (b), the constraint supports or counteracts the objective: $\Pr(\sum_{i=1}^{N} \xi_i x_i \odot v) \geq p$. The final bounds on the optimal objective value are of the form $\underline{\omega}^{\odot} + \underline{\omega}^{\otimes} \leq \hat{\omega} \leq \overline{\omega}^{\odot} + \overline{\omega}^{\otimes}$. An entry with — means that no bound exists for that case. The final bound is the best bound from this table where any of the cases are true. For example, for $\underline{\omega}^{\odot}$, the final bound is the *maximum* of $p\underline{s}\underline{l}$ (or $\underline{s}\overline{l}$) and pv (or v). Notice how there is always at least one available bound from group (a), and possibly one additional bound from group (b), so that at least one bound for each column of the table always exists.

Proof. Consider a counteracting probabilistic constraint $\Pr(\sum_{i=1}^{N} \xi_i x_i \odot v) \geq p$. We prove that $\hat{\omega} \geq \underline{\omega}^{\odot} + \underline{\omega}^{\otimes}$ under condition $v \geq 0$, which was mentioned in Section 4.6.4 for a minimization objective with a counteracting constraint. Following the table for case $v \geq 0$, we have to prove $\hat{\omega} \geq pv$:

$$\begin{aligned}\hat{\omega} &= \hat{\omega}^{\odot} + \hat{\omega}^{\otimes} = \frac{1}{\hat{M}} \sum_{j \in \hat{S}_{\hat{x}}^{\odot}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\ \hat{x}_i + \frac{1}{\hat{M}} \sum_{j \in \hat{S}_{\hat{x}}^{\otimes}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\ \hat{x}_i \\ &\geq \frac{1}{\hat{M}} \sum_{j \in \hat{S}_{\hat{x}}^{\odot}} v = \frac{1}{\hat{M}} |\hat{S}_{\hat{x}}^{\odot}| v \geq pv,\end{aligned}$$

where we used the facts that $|\hat{S}_{\hat{x}}^{\otimes}| \geq 0$ and $|\hat{S}_{\hat{x}}^{\odot}| \geq p\hat{M}$. We now prove that $\hat{\omega} \leq \overline{\omega}^{\odot} + \overline{\omega}^{\otimes}$ under conditions $\overline{s} \geq 0$ and $v \geq 0$, which was also mentioned in Section 4.6.4 for a minimization objective with supporting constraint. Following the table, one possible such bound under these conditions is $\hat{\omega} \leq v + (1-p)\overline{s}\overline{l}$, which follows from:

$$\begin{aligned}\hat{\omega} &= \hat{\omega}^{\odot} + \hat{\omega}^{\otimes} = \frac{1}{\hat{M}} \sum_{j \in \hat{S}_{\hat{x}}^{\odot}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\ \hat{x}_i + \frac{1}{\hat{M}} \sum_{j \in \hat{S}_{\hat{x}}^{\otimes}} \sum_{i=1}^{N} \hat{s}_{ij}.\mathsf{A}\ \hat{x}_i \\ &\leq \frac{1}{\hat{M}} \sum_{j \in \hat{S}_{\hat{x}}^{\odot}} v + \frac{1}{\hat{M}} \sum_{j \in \hat{S}_{\hat{x}}^{\otimes}} \sum_{i=1}^{N} \overline{s}\ \hat{x}_i \leq \frac{1}{\hat{M}} |\hat{S}_{\hat{x}}^{\odot}| v + \frac{1}{\hat{M}} |\hat{S}_{\hat{x}}^{\otimes}| \overline{s}\overline{l} \\ &\leq v + (1-p)\overline{s}\overline{l},\end{aligned}$$

where we used the facts that $|\hat{S}_{\hat{x}}^{\odot}| \leq \hat{M}$ and $|\hat{S}_{\hat{x}}^{\otimes}| \leq (1-p)\hat{M}$. All other bounds in the table are easily derivable following a similar approach. ■

4.6.5 Implementation Considerations

We now discuss several implementation optimizations.

Efficient summary generation. Recall that summarization has two steps: (1) computing the scenario scores to sort scenarios by the previous solution, and (2) computing the

tuple-wise minimum (or maximum) of the first $\alpha\%$ of the scenarios in sorted order. The fastest way to generate an α-summary is if all M scenarios are generated and kept in main memory at all times. In this case, computing the tuple-wise minimum (or maximum) is trivial. However, the $\Theta(MNK)$ memory requirement for this may exceed the memory limits if M is large. We devise two possible strategies for memory-efficient summary generation with optimal $\Theta(NZK)$ space complexity: *tuple-wise summarization* and *scenario-wise summarization*. Tuple-wise summarization uses a unique random number seed for each tuple ($i = 1, \ldots, N$) and it generates all M realizations, one tuple at a time. Scenario-wise summarization uses a unique seed for each scenario ($j = 1, \ldots, M$), and it generates one realization for all tuples, one scenario at a time.

With tuple-wise summarization, sorting the scenario only requires $\Theta(PM)$ time, where $P = \sum_{i=1}^{N} x_i$ is the size of the current package; usually, $P \ll N$. However, generating the summaries is more costly, as it requires $\Theta(NM)$ time, as all M realizations must be constructed for all N tuples. The total time is $\Theta(M(P+N))$. With scenario-wise summarization, generating summaries has lower time complexity of $\Theta(\alpha NM)$, as it only generates scenarios in $G_z(\alpha)$, but sorting has higher complexity $\Theta(NM)$, with total time $\Theta(NM(\alpha+1))$.

It follows that if $\alpha \geq P/N$, tuple-wise summarization is generally faster than scenario-wise summarization. However, other factors may affect the runtime, e.g., some random number generators, such as Numpy, generate large quantities of random numbers faster if generated in bulk using a single seed. In this case, tuple-wise summarization may suffer considerably in the summary generation phase, as it needs to re-seed the random number generator for each tuple. In our experiments, we observed that tuple-wise summarization is better when the input table is relatively small, but worse than scenario-wise for larger tables. In general, a system should implement both methods and test the two in situ.

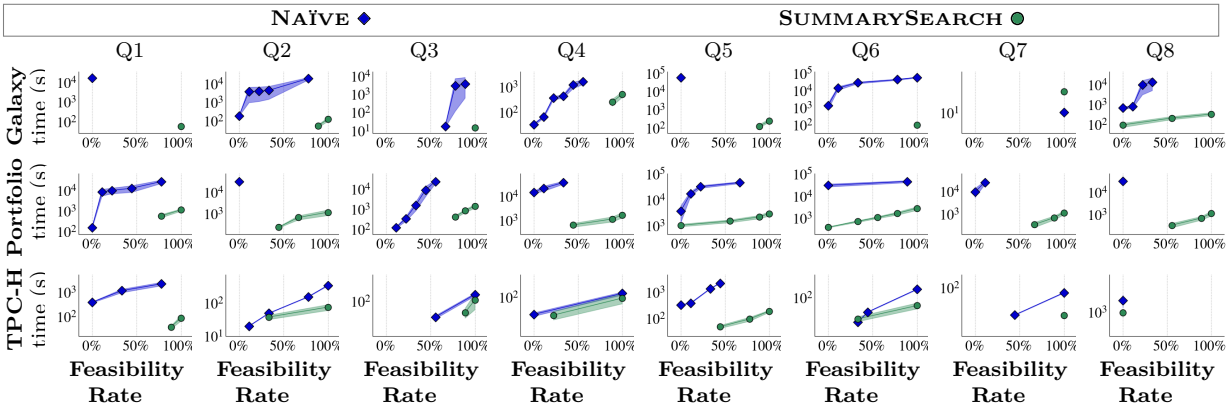


Figure 4.4: End-to-end results of SUMMARYSEARCH vs. NAÏVE. Plotting the average time (and 95% confidence intervals) to reach 100% feasibility rate. Of the 23 feasible queries (TPC-H Q8 is infeasible), SUMMARYSEARCH always reaches 100% feasibility rate, while NAÏVE in only 7 queries. In 15 queries, when SUMMARYSEARCH succeeds, NAÏVE is still at 0% feasibility. SUMMARYSEARCH can be orders of magnitude faster even when both reach 100% feasibility.

Convergence acceleration. When $\alpha_k^{(q)}$ is obtained by *decreasing* $\alpha_k^{(q-1)}$, the solution $x^{(q-1)}$ typically is feasible, and our goal is for $x^{(q)}$ to strictly improve in objective value. CSA-SOLVE achieves this by slightly modifying the generation of summaries in order to ensure that the previous solution is still feasible for the next CSA problem. This is done by using the tuple-wise maximum (instead of minimum) in the summary generation for all tuples t_i such that $x_i^{(q-1)} > 0$ (tuples in the previous solution). For all other tuples, we set the summary as usual. We have found that ensuring monotonicity of the objective values promotes faster convergence.

4.7 Experimental evaluation

In this section, we present an experimental evaluation of our techniques for stochastic package queries on three different domains where uncertainty naturally arises: noise in sensor data, uncertainty in future predictions, uncertainty due to data integration [53]. Our

results show that: (1) **SummarySearch** is always able to find feasible solutions, while **Naïve** cannot in most cases—when both **SummarySearch** and **Naïve** can find feasible solutions, **SummarySearch** is often faster by orders of magnitude; (2) The packages produced by **SummarySearch** are of high quality (low empirical approximation ratio), sometimes even better than **Naïve** when they both produce feasible solutions; (3) Increasing M, the number of optimization scenarios, helps **SummarySearch** find feasible solutions, and the value of M required by **SummarySearch** to start producing feasible solutions is much smaller than **Naïve**, explaining the orders of magnitude improvement in running time; (4) Increasing Z, the number of summaries, helps **SummarySearch** find higher-quality solutions; (5) Increasing N, the number of input tuples, impacts the running time of both algorithms, but **SummarySearch** is still orders of magnitude faster than **Naïve**, and finds feasible solutions with better empirical approximation ratios than **Naïve**.

4.7.1 Experimental setup

We now describe the software and runtime environment, and the three workloads we used in the experiments.

Environment. We implemented our methods in Python 2.7, used Postgres 9.3.9 as the underlining DBMS, and IBM CPLEX 12.6 as the ILP solver. We ran our experiments on servers equipped with two 24 2.66GHz cores, 15GB or RAM, and a 7200 RPM 500GB hard drive.

Datasets and queries. We constructed three workloads:

Noisy sensor measurements: The Galaxy datasets vary between 55,000 and 274,000 tuples, extracted from the Sloan Digital Sky Survey (SDSS) [152], with different queries using

subsets of the original Galaxy dataset (Section 3.4.1) of different sizes. Each tuple contains the color components of a small portion of the sky as read by a telescope. We model the uncertainty in the telescope readings as Gaussian or Pareto noise.

Financial predictions: The Portfolio dataset contains $6,895$ stocks downloaded from Yahoo Finance [165]. The initial price of each stock is set according to its actual value on January 2, 2018, and future prices are generated according to a geometric Brownian motion. Following Figure 4.1, our Portfolio queries serve an investor who wants to buy a set of shares "today" (in our case, January 2, 2018), using predictions for the next few days. We construct two datasets, with different prediction horizons: in the *short-term* (resp., *long-term*) dataset, stocks can be sold back to the stock market at most two (resp., seven) days from now. The dataset for the short-term (resp., long-term) trades contains $14,000$ (resp., $48,000$) tuples. For each of these two types, we also extracted subsets corresponding to the 30% most *volatile* stocks to construct some of the hardest queries, where volatility is defined as the standard deviation of the past stock's prices. Different queries refer to datasets of different horizons and volatility selection. Tuples referring to the same stock are correlated to one another. For example, in Figure 4.1, tuples 1 and 2 are correlated to each other but are independent of the other tuples.

Data integration: The TPC-H dataset consists of about 117,600 tuples extracted from the TPC-H benchmark [153]. We simulate the result of hypothetically integrating several data sources to form this data set: we model uncertainty in each attribute's value with discrete probability distributions. For each original (deterministic) value in the TPC-H dataset, we generate D possible variations thereof, where D is the number of data sources that have been integrated into one. The mean of these D values is anchored around the original value; each source value is sampled from an exponential, Poisson, uniform or Student's t-distribution.

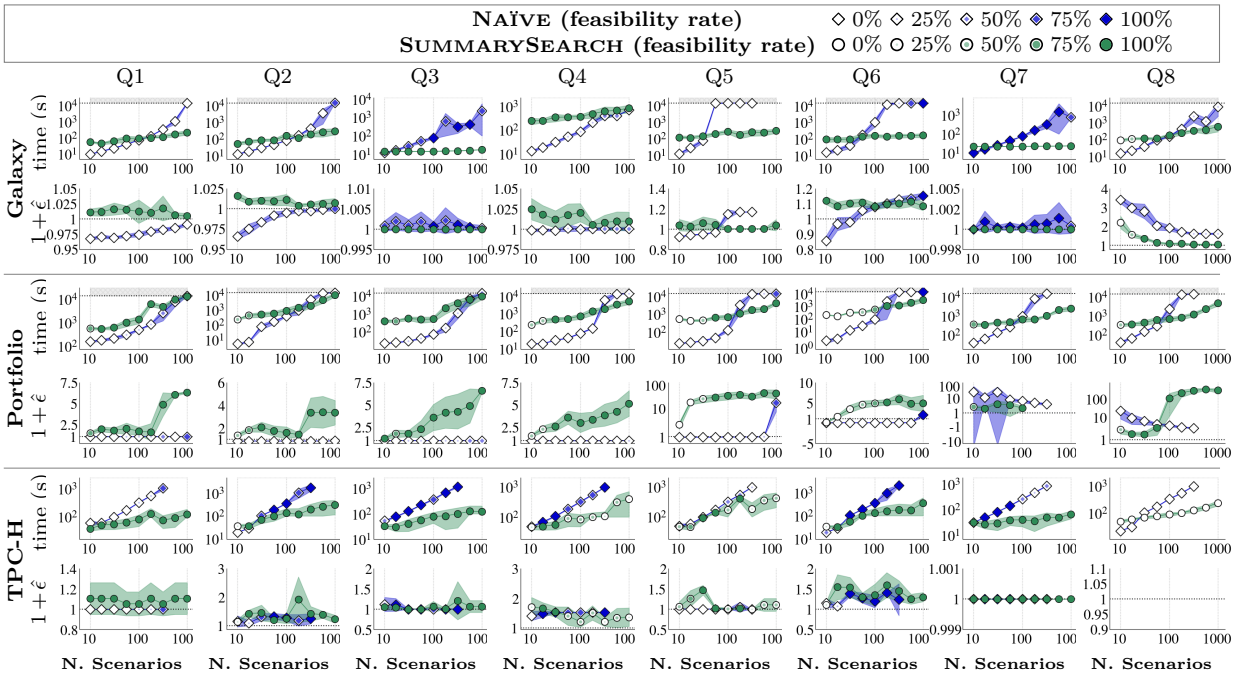


Figure 4.5: Scalability of NAÏVE and SUMMARYSEARCH with increasing number of optimization scenarios. NAÏVE struggles to find feasible solutions even with a large number of scenarios and often fails completely (missing points in the plot). SUMMARYSEARCH quickly finds feasible solutions with few scenarios. The approximation ratios of SUMMARYSEARCH's solutions are generally low when the number of scenarios is small.

Galaxy query template (counteracted objective)
SELECT PACKAGE(∗) FROM Galaxy SUCH THAT
  COUNT(∗) BETWEEN 5 AND 10 AND
  SUM(Petromag_r) ≥ **{v}** WITH PROBABILITY ≥ **{p}**
MINIMIZE EXPECTED SUM(Petromag_r)

Galaxy query template (supported objective)
SELECT PACKAGE(∗) FROM Galaxy SUCH THAT
  COUNT(∗) BETWEEN 5 AND 10 AND
  SUM(Petromag_r) ≤ **{v}** WITH PROBABILITY ≥ **{p}**
MINIMIZE EXPECTED SUM(Petromag_r)

Portfolio query template (supported objective)
SELECT PACKAGE(∗) FROM Stock_Investments SUCH THAT
  SUM(price) ≤ 1000 AND
  SUM(Gain) ≥ **{v}** WITH PROBABILITY ≥ **{p}**
MAXIMIZE EXPECTED SUM(Gain)

TPC-H query template (independent objective)
SELECT PACKAGE(∗) FROM Tpch_**{D}** SUCH THAT
  COUNT(∗) BETWEEN 1 AND 10 AND
  SUM(Quantity) ≤ **{v}** WITH PROBABILITY ≥ **{p}**
MAXIMIZE PROBABILITY OF SUM(Revenue) ≥ 1000

Figure 4.6: Query templates for the three workloads used in the experimental evaluation of SummarySearch and Naïve. Each parameter in a template is indicated in curly brackets.

For each of the three datasets, we constructed a workload of eight sPaQL queries; all 24 queries, except one in TPC-H, are feasible. The workloads span seven different distributions for the uncertain data attributes, including a complex VG function to predict future stock prices. The objective functions are supported by the constraints for the Portfolio queries, independent for the TPC-H queries and either supported or counteracted for the Galaxy queries (see Definition 11 for supported/counteracted/independent objectives). The Portfolio workload tests high- and low-risk, high- and low-VaR (Value at Risk)—i.e., p and v in Equation (4.1)—as well as short- and long-term trade predictions. The TPC-H workload is split into queries with $D = 3$ and $D = 10$ (number of integrated sources). For all queries there are two constraints, one of which is probabilistic with $p \geq 0.9$. Examples

include: (1) for Galaxy, we seek a set of five to ten sky regions that minimizes total expected radiation flux while avoiding total flux levels higher than 40 with high probability, and (2) for TPC-H, we seek a set of between one and ten transactions having maximum expected total revenue, while containing less than 15 items total with high probability. Figure 4.6 shows the sPaQL query templates for each dataset. The parameters in the templates are indicated under curly brackets. Table 4.3 provides all the remaining details for the datasets and queries, including all the query parameters.

Evaluation metrics. We measure *response time* (in seconds and logarithmic scale) across 10 i.i.d runs using different seeds for generating the optimization scenarios, and evaluate feasibility and the objective value on an out-of-sample validation set with 10^6 scenarios (10^7 for the Portfolio workload). We plot the average across the 10 runs, and its 95% confidence interval in a shaded area. For each run of an algorithm, we set a time limit of four hours. When the time limit expires, we interrupt CPLEX and get the best solution found by the solver until then. We measure *feasibility rate* as the fraction, out of the 10 runs, in which a method produces a feasible solution (including, for all methods, when the time limit expired). Because the true optimal solution for any of the queries is unknown, we measure *accuracy* by $1+\hat{\epsilon}$, where $\hat{\epsilon} := \omega/\omega^* - 1$ and ω^* is the objective value of the best feasible solution found by any of the methods.

4.7.2 Results and discussion

We evaluate four fundamental aspects of our algorithms: (1) query response time to reach 100% feasibility rate; (2) scalability with increasing number of scenarios (M); (3) scalability of SummarySearch with increasing number of summaries (Z); (4) scalability with increasing dataset size (N).

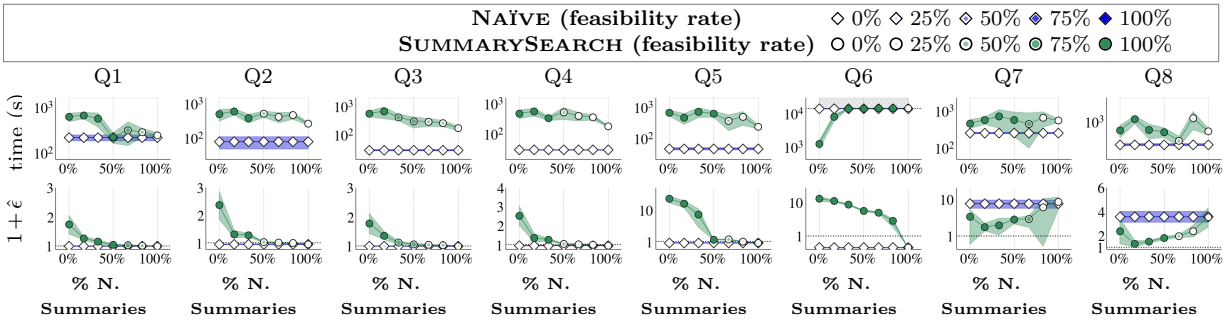


Figure 4.7: Effects of increasing number of summaries (Z) on the Portfolio workload, as a percentage of the number of scenarios, from 1 summary up to M summaries (100%). Increasing Z improves the approximation ratio of the solution produced by SUMMARYSEARCH. Increasing Z too far results in infeasible solutions as, when $Z = M$, SUMMARYSEARCH is identical to NAÏVE, and it thus overfits, like NAÏVE, to a bad set of scenarios.

4.7.2.1 Response time to reach 100% feasibility rate

Both **NAÏVE** and **SUMMARYSEARCH** increase M (the number of scenarios) up to when solutions start to be feasible. We report the cumulative time for all iterations the algorithm took to reach a certain feasibility rate, from 0%, up to 100% (when the algorithm produces feasible solutions for all 10 runs). For **SUMMARYSEARCH**, Z is fixed (1 for Galaxy and Portfolio, 2 for TPC-H). We set Z to the lowest value (per workload) such that **SUMMARYSEARCH** could reach 100% feasibility rate. Figure 4.4 shows the results of the experiment. For all (23) feasible queries across all workloads, **SUMMARYSEARCH** is always able to reach 100% feasibility rate, while **NAÏVE** can only reach 100% feasibility for only 7 queries. Even then, **SUMMARYSEARCH** is usually orders of magnitude faster than **NAÏVE** (e.g., Galaxy Q6, TPC-H Q2, Q6, and Q7). Moreover, in 15 out of the 23 feasible queries, **SUMMARYSEARCH** reached 100% feasibility while **NAÏVE** was still at 0%. The conservative nature of summaries allows higher feasibility rates for **SUMMARYSEARCH** even with fewer scenarios. As the number of scenarios increases, **SUMMARYSEARCH** solves a much smaller problem than **NAÏVE**, leading to orders-of-magnitude faster response time.

The only case where **SummarySearch** is slower than **Naïve** at reaching 100% feasibility rate is Galaxy Q7, which was an easy query for both methods: both solved it with only 10 scenarios. This query has a supported objective function over data with minimal uncertainty described by a Pareto distribution with "scale" and "shape" both equal to 1. For this query, the summarization process and solving a probabilistically-unconstrained problem are overheads for **SummarySearch**. TPC-H Q8 is an infeasible query. Both methods increase M up to 1000 before declaring infeasibility, but again **SummarySearch** is faster than **Naïve** in doing so.

4.7.2.2 Effect of increasing the number of optimization scenarios

We evaluate the scalability of our methods when the number of optimization scenarios M increases; Z is fixed as described above. For each algorithm, we group feasibility rates into 5 groups: 0%, 25%, 50%, 75% and 100%, and use different shadings to distinguish each case.

Figure 4.5 gives scalability results for the three workloads. Generally, with low M, **Naïve** executes very quickly to produce infeasible solutions with low objective values (optimizer's curse); as **Naïve** increases M, the running time increases exponentially—note the logarithmic scale—up to a point where it fails altogether (missing **Naïve** points in the plots). On the other hand, **SummarySearch** finds feasible solutions even with as little as 10 scenarios.

SummarySearch produces high quality solutions as demonstrated by the low approximation ratio $(1+\hat{\epsilon})$, close to 1 for most queries. However, with the hardest Portfolio queries (Q5 and Q6), the worst approximation ratio for **SummarySearch** is quite high for feasible solutions: this is an indicator that the number of summaries, $Z = 1$ is too low and should be increased.

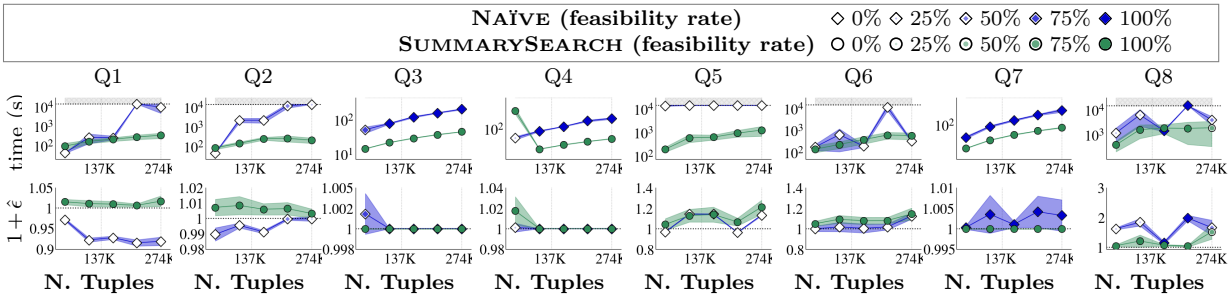


Figure 4.8: Scalability of NAÏVE and SUMMARYSEARCH with increasing dataset size (N) on the Galaxy workload. The running times of both algorithms degrades with increasing N, but SUMMARYSEARCH scales up well in comparison with NAÏVE.

4.7.2.3 Effect of increasing the number of summaries

In this experiment, we show how increasing the number of summaries (Z) helps improve the approximation ratio in the Portfolio queries. We increase Z from 1 up to M (number of scenarios), where M is set to where the feasibility rate of **SUMMARYSEARCH** was 100% in the previous experiment, and we show the running time and approximation ratio compared to **NAÏVE** with M scenarios. Figure 4.7 shows the results of this experiment. First, the response time with increasing Z is in most cases independent of Z. In fact, while increasing Z adds more scenarios to the CSA formulation, each summary becomes less and less conservative, making the problem a bit larger but always easier; in the limit ($Z = M$), each summary is identical to an original scenario, and thus **SUMMARYSEARCH** only pays the extra overhead, compared to **NAÏVE**, of solving the probabilistically-unconstrained problem first. On the other hand, **NAÏVE** is always faster, but its solutions are infeasible. For most queries, the approximation ratio closely approaches 1, while still maintaining a high feasibility rate. Increasing Z too far eventually causes feasibility to drop, reaching that of **NAÏVE** in the limit ($Z = M$).

Finally, even though infeasible solutions tend to have better objective values than feasible ones, we find that **Naïve**'s infeasible solutions to Q7 and Q8 have worse objective values. These queries proved quite challenging for **Naïve** as they involved stock price predictions for a week in the future.

4.7.2.4 Effect of increasing the dataset size

In this experiment, presented in Figure 4.8, we increase the Galaxy dataset up to five times from 55,000 tuples to 274,000 tuples. For all queries except Q8 we fix $M = 56$ (for both algorithms) and $Z = 1$. In general, **SummarySearch** scales well with increasing data set size: it finds feasible solutions with good approximation ratios. **Naïve**, however, times out for several queries (Q1, Q2, Q5, Q6, & Q8) and its response time sharply increases as dataset size increases (Q1, Q2, Q6, Q8). Except for three queries (Q3, Q4, Q7), most of **Naïve**'s solutions are infeasible; even then, **SummarySearch** produces feasible solutions in orders of magnitude less time with better approximation ratios.

In Q8, we set $M = 562$ to enable **SummarySearch** to still produce feasible solutions (75% feasibility at 274K tuples), without causing **Naïve** to fail. Q8 is a challenging query as each data value is sampled from a Pareto distribution with different parameters leading to high variability across scenarios.

To further increase the data size scalability of **SummarySearch**, we hope to combine it with partitioning and divide-and-conquer approaches similar to **SketchRefine**.

4.8 sPaQLTooLs: A sPaQL interface for stochastic constrained optimization

In this section, we describe sPaQLTooLs, our interface for sPaQL queries that employs **SummarySearch** for quickly producing feasible and close-to-optimal package solutions. Recall that **SummarySearch** approximates the given stochastic ILP (SILP) by a deterministic ILP (DILP) that simultaneously incorporates multiple "*scenarios*", or possible worlds, for the future stock market. A scenario is a table where all random variables have been realized. The right-hand side of Figure 4.1 shows a possible scenario. To generate scenarios, we employ the Monte Carlo probabilistic data model [80], which offers support for arbitrary distributions via user-defined *variable generation* (VG) functions.

The solution of the DILP, however, may not be feasible with respect to the original SILP, especially if the approximation is based on only a small number of scenarios that do not well represent the true uncertainty distribution. For example, a financial package obtained by using too few scenarios might have a 10% probability of losing more than \$10, rather than a 5% probability, incurring more risk than desired. The state-of-the-art techniques attempt to mitigate this by iteratively adding more scenarios into the DILP. We implemented this approach in an algorithm that we call **Naïve**; unfortunately, the **Naïve** DILP may quickly become too large for the solver to handle, and this approach often fails.

Recall that our approach, **SummarySearch**, instead facilitates feasible packages by replacing a set of scenarios with a very small synopsis thereof, called a "summary", which results in a "reduced" DILP that is much smaller than the original DILP used by **Naïve**. A summary is carefully crafted to be "conservative" in that the constraints in the reduced DILP are harder to satisfy than the constraints in the **Naïve** DILP. Because the reduced DILP is much smaller than the **Naïve** DILP, it can be solved much faster; moreover,

the resulting solution is much more likely to be feasible, so that the required number of iterations is typically reduced. Of course, if a summary is overly conservative, the resulting solution will be feasible, but highly suboptimal. Therefore, during each optimization phase, **SummarySearch** implements a sophisticated search procedure aimed at finding a "minimally" conservative summary; this search requires solution of a sequence of reduced DILPs, but each can be solved quickly.

With our interface, sPaQLTooLs [147], shown in Figure 4.9, users can easily construct a Financial Portfolio using **SummarySearch** underneath. While we show how sPaQLTooLs can help build investment plans, sPaQLTooLs is *generic* in that it can be easily configured to support applications of constrained optimization with uncertainty other than the Financial Portfolio.

In the interface, users indicate their investment budget to construct an investment portfolio on real stock market data. They can first attempt to build a Financial Portfolio *manually*, using common-sense techniques, such as looking for low-volatility stocks, and greedily adding one stock at a time to the portfolio package. The system then evaluates their manually-constructed portfolio on a large number of scenarios. Manually-constructed portfolios are unlikely to be feasible, and therefore users can experience first-hand the difficulty of building low-risk portfolios without our automated methods for sPaQL queries. Finally, the user can easily solve the financial problem *automatically*, as a sPaQL query, via our advanced **SummarySearch**. The interface shows how the initial solutions found by the system can potentially also be infeasible, and how quickly the system finds feasible solutions and improve on their objective value (expected gain). We also compare the solutions found by **SummarySearch** and (when possible) the **Naïve** algorithm.

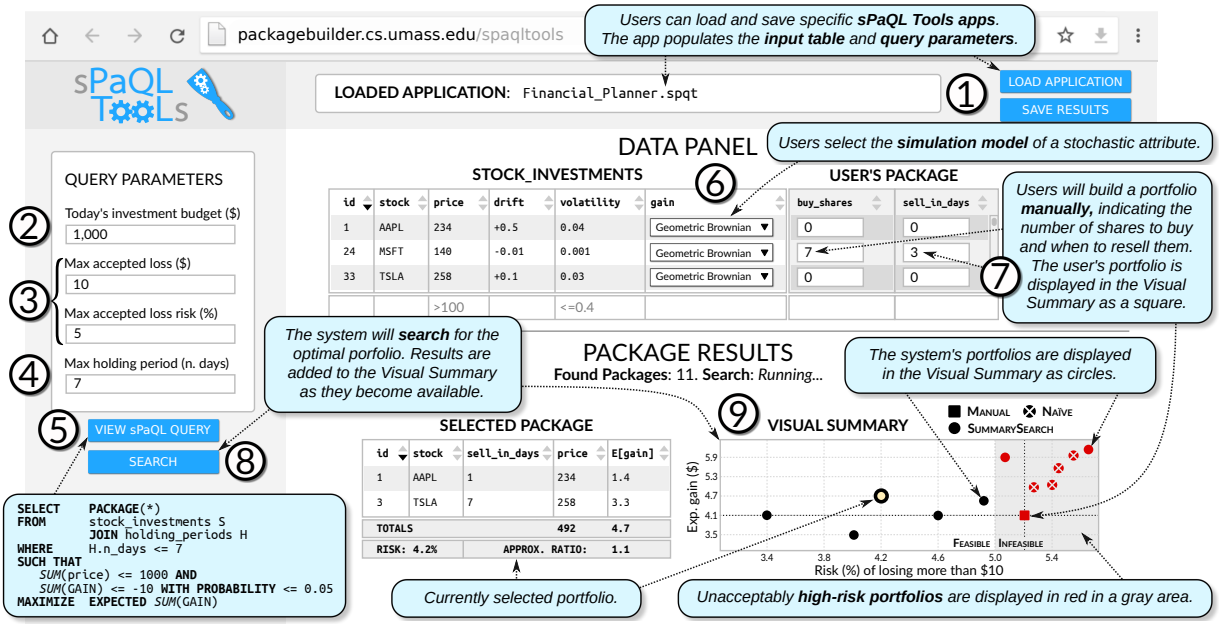


Figure 4.9: sPaQLTooLs GUI and demo outline: ① application loading, ②③④ query specification, ⑤ sPaQL inspection, ⑥ simulation modeling, ⑦ manual solution, ⑧ automatic solutions, ⑨ result exploration.

4.8.1 Example usage scenario

We showcase our sPaQL query engine on a real dataset of stocks for portfolio optimization. Figure 4.9 shows a screenshot of the sPaQLTooLs's graphical user interface. During the interaction with the interface, a user typically goes through the following steps.

Step ① (Loading the sPaQLTooLs application). The user loads a sPaQLTooLs application from a `.spqt` configuration file, which points to the input database and tables, as well as the sPaQL query template and input parameters that populate all the graphical interface elements. While the interface can support a variety of applications, we showcase the Financial Portfolio. This phase populates the `Stock_Investments` table and the query parameters described next.

Step ② (Specifying the budget constraint). The first query parameter indicates how much money ($) the user wants to invest at most.

Step ③ (Specifying the accepted risk constraint). The next two parameters set the risk level the user is willing to accept with the investment. The maximum accepted loss indicates how much money ($) the user is willing to lose at most, and the risk (%) indicates the maximum probability with which this loss can occur.

Step ④ (Specifying the maximum holding period). The user then specifies the maximum number of days d to hold any stock. This will generate d rows in the `Stock_Investments` table for each unique stock. The larger the value of d, the more uncertainty and hence the harder the optimization.

Step ⑤ (sPaQL inspection). Once all query parameters have been specified, the user can inspect the sPaQL query used by the system to search for the optimal portfolio.

Step ⑥ (Specifying the simulation model). For all the stochastic attributes in the input table (in this application, only Gain), the user selects a simulation model from a drop-down menu. Users can select a different simulation model for different tuples, the same for all tuples, or any other combination. Simulation models are defined in the application configuration.

Step ⑦ (Manual financial planning). Users can build a Financial Portfolio manually, in an attempt to compete against our system that uses **SummarySearch** to find the optimal portfolio automatically. For example, a user may filter stocks by their volatility and price, as shown in the figure, and decide to buy a number of shares for some low-volatility stocks. As the user starts building their manual portfolio, the system runs simulations in the background, to estimate the associated risk and expected gain. The constructed portfolio is then shown in the Visual Summary at the bottom as a square,

placed in the graph according to its risk and expected gain. The portfolio may be *infeasible*, i.e., it may not not satisfy the risk constraint, in which case it is colored in red and placed in a gray area. Users experience first-hand the difficulty of manually constructing feasible portfolios with high-enough gain.

Step ⑧ (Automatic financial planning). Users then run our system to search for the optimal portfolio according to their needs. We concurrently run both **SummarySearch** and **Naïve** (for comparison).

Step ⑨ (Visual exploration of the results). The results of our system are interactively shown in the Visual Summary as they become available. As the system produces a solution, the interface plots its associated risk and expected gain. As feasible solutions are found, **SummarySearch** starts improving their objective value (expected gain). At the end of the search, the final solution has the highest expected gain, under the acceptable risk. Users are able to compare this result with the portfolio that they manually built. They can also click on any solution in the Visual Summary in order to view the full portfolio details, as a table on the left of the summary.

4.9 Conclusion

In this chapter, we addressed *single-stage* decision making under uncertainty, in which decisions are made before the values of the random variables become known. In many cases, however, uncertainty is revealed over time, in stages, allowing for remedial actions. We plan to explore these dynamic settings, referred to as stochastic programming with recourse. Another goal is to extend our methods to problems that involve probabilistic constraints where the inner constraints must *jointly* be satisfied with a given probability; such an extension is highly nonntrivial. We also plan to work on further algorithmic improvements,

including (i) developing more sophisticated summarization methods than minimum and maximum summaries; (ii) scaling up **SummarySearch** to very large datasets (e.g., millions of tuples) by combining summaries with divide-and-conquer approaches like **SketchRefine** [23]; (iii) parallelizing CSA-Solve and summary generation; and (iv) fully integrating stochastic package queries into a probabilistic database to handle multi-table queries. We plan to further develop our theory on **SummarySearch** to formally prove its convergence to feasible solutions as the number of scenarios increases. Finally, we plan to explore ways to "open the black box" of optimization software to allow for further performance improvements, in analogy to the way MCDB re-engineered query operations to efficiently handle uncertain tuple attributes. Other limitations of **SummarySearch** and future research directions are discussed in Chapter 5, Section 5.1.

	Dataset		Query					
	N	Uncertainty		Feasible?	Objective	Supportiveness	p	v	Other features
Galaxy	55,000 to 274,000	Normal(σ=2)	Q1	Yes	$\min \mathbb{E}(\cdot)$	Counteracted	0.9	40	
		Normal(σ^*=3)	Q2					43	
		Normal(σ=2)	Q3			Supported		50	
		Normal(σ^*=3)	Q4					52	
		Pareto(σ=α=1)	Q5			Counteracted		65	
		Pareto(σ^*=α=1)	Q6					65	
		Pareto(σ=α=1)	Q7			Supported		109	
		Pareto(σ^*=3, α=1)	Q8					90	
Portfolio	4,000 to 14,000	Geometric Brownian Motion	Q1	Yes	$\max \mathbb{E}(\cdot)$	Supported	0.9	−10	2-day, All stocks
			Q2				0.95	−10	2-day, All stocks
			Q3				0.9	−10	2-day, Most volatile
			Q4				0.95	−10	2-day, Most volatile
			Q5				0.9	−1	2-day, Most volatile
			Q6				0.95	−1	2-day, Most volatile
			Q7				0.9	−10	1-week, Most volatile
			Q8				0.9	−1	1-week, Most volatile
TPC-H	117,600	Exponential(λ=1)	Q1	Yes	$\max \Pr(\cdot)$	Independent	0.9	15	D=3
		Exponential(λ=1)	Q2				0.95	7	D=10
		Poisson(λ=2)	Q3				0.9	15	D=3
		Poisson(λ=1)	Q4				0.9	10	D=10
		Uniform(0,1)	Q5				0.9	15	D=3
		Uniform(0,1)	Q6				0.95	7	D=10
		Student's t(ν=2)	Q7				0.9	29	D=3
		Student's t(ν=2)	Q8	No			0.95	7	D=10

Table 4.3: Detailed description of datasets and queries. For Galaxy, the means of the distributions are always the original data values, and we thus only indicate the other distribution parameters (standard deviation σ and shape α); The standard deviations can be of two kinds: all identical (indicated by σ), or all different and randomly generated (indicated by σ^*); In the second case, the standard deviations of the tuples were generated randomly using a normal distribution with mean zero and standard deviation σ^*, and by then taking their absolute values. For Portfolio, "2-day" indicates predictions made only for the following two days, and "1-week" indicates predictions for an entire week; "Most volatile" indicates that the dataset only includes the 30% most volatile stocks. For TPC-H, we indicate the distribution used to model the data integration uncertainty, and D, the number of integrated sources.

CHAPTER 5

LIMITATIONS AND FUTURE DIRECTIONS

While we believe that the techniques presented in this thesis are very powerful, they have several limitations. This chapter outlines some of these limitations, as well as research directions for addressing them. Further, it identifies two key challenging areas related to package queries that deserve full attention for future work: package queries by example (PQBE) and incremental package evaluation (IPE). The chapter includes preliminary results to help identify the major challenges that these two areas present. The chapter concludes with our vision on data management systems for data-driven decision making.

5.1 Limitations

In this section, we outline the major limitations of the solutions presented in this thesis, and discuss research directions for addressing them.

5.1.1 Robustness of SketchRefine

SketchRefine [59, 25, 23], presented in Chapter 3, is able to scale computation of package queries on very large datasets that ILP solvers cannot handle directly. This is achieved by breaking down the input dataset into several disjunct partitions, each containing tuples that are similar enough to each other so that they can be replaced by a single representative. Thus, the original dataset is reduced by several orders of magnitude, so that the resulting ILP is small enough for a solver to handle efficiently. This step, called

Sketch, produces an initial solution quickly, which can be later refined, in the **Refine** step, issuing appropriate ILPs that each replace a representative with some of the original real tuples. **SketchRefine** ensures that if the dataset partitioning follows certain strict *quality criteria* (see Section 3.3.2), then any solution for any package query will have strong approximation guarantees, governed by a user-defined parameter ϵ (the smaller ϵ, the better the guarantee). Further, our empirical results show that a "generic" partitioning, i.e., one which does not enforce those quality criteria, still yields solutions with very good quality in all datasets and queries we tested. However, there are still two major limitations of **SketchRefine**: (1) there is no guarantee that the quality criteria *can* be achieved for *any* input dataset, and (2) good empirical performance without those criteria is not guaranteed, as there could be datasets for which **SketchRefine** can perform poorly. Here, we describe these two limitations more in detail, and the future directions to address them.

The quality criteria required by **SketchRefine** to achieve an approximation guarantee might be too strict to be achieved, especially on very skewed datasets and/or when the ϵ required by the user is too small. One way **SketchRefine** can cope with this issue is the following. During partitioning, it *tries* to achieve the desired quality criteria; if, at some point, the procedure realizes that the criteria cannot be achieved, it stops the partitioning procedure prematurely, and informs the user of the best ϵ-guarantee that the current partitioning can offer. The user can then decide whether that ϵ is good enough for their application. Further research is necessary to offer other alternatives to the user, should the best possible ϵ be too large.

While in all our experiments, we have seen good performance of **SketchRefine** that uses a generic partitioning, there might be datasets and queries where the algorithm performs poorly unless the partitioning is constructed with the quality criteria. A sufficient

condition for this bad behavior to occur is having a partitioning where some of the representatives are very far away from their represented tuples, and **Sketch** chooses exactly those representatives to form the initial solution. In a case like this, solutions are likely to be very suboptimal. A simple approach to shield **SketchRefine** from this is to re-partition *online* (at runtime) in case **Sketch** chooses bad partitions. After re-partitioning, new representatives are added to the **Sketch** dataset, which can help identify better solutions. This approach is simple, but the downsides are that (1) it does not offer the same guarantees as the original **SketchRefine** approach, and (2) it incurs extra runtime costs. Further research is necessary to study how much this can help improve the quality of the solutions, and to find other alternatives that can save runtime while providing protection against bad partitionings.

5.1.2 Stochastic package queries on very large datasets

SketchRefine and **SummarySearch** solve two orthogonal problems: the former deals with deterministic package optimization on very large tables; **SummarySearch** [28] solves stochastic optimization that require too many scenarios. While the need for too many scenarios is a direct consequence of an increased data set size, it can typically happen even with relatively small sizes. When the data set size explodes, stochastic optimization becomes even more prohibitively expensive, and **SummarySearch** alone may not be sufficient any more. A possible solution to this issue might be to combine the capability of **SketchRefine** to deal with very large data sizes with the capability of **SummarySearch** to deal with the uncertainty of large datasets, into a *hybrid* new algorithm. Since both **SketchRefine** and **SummarySearch** are complex algorithms, devising a combined algorithm is naturally challenging.

5.1.3 Stochastic package queries with joint probabilistic constraints

The methods developed in this thesis support *individual* probabilistic constraints. However, many stochastic problems require several constraints to be satisfied *jointly* with some probability. While it is easy to augment our language to support these constraints, computing solutions for package queries with joint probabilistic constraints is more challenging as they generalize individual probabilistic constraints. A possible avenue is to explore the applicability of existing techniques from the stochastic programming literature to the context of large tables, and to extend **SummarySearch** with summaries that are specifically tailored for joint constraints.

5.1.4 Multi-stage stochastic packages

This thesis addressed what is referred to as *single-stage* decision making under uncertainty, in which decisions have to be made before the values of the random variables become known. However, many applications require uncertainty to be revealed over time, i.e., in stages, allowing for remedial actions. These dynamic settings, also referred to as *stochastic programming with recourse*, are more challenging to address than the single-stage setting. An important extension of the work presented in this thesis is to study how to solve these complex problems at a large scale.

5.1.5 Deep implementation in a relational database system

The system presented in this thesis sits on top of an existing DBMS. While this design choice substantially simplifies the implementation of the system and allows several different DBMSs to be used transparently underneath, it potentially prevents solutions that can leverage a deeper integration *inside* of a specific DBMS. Putting the solver inside the DBMS is more challenging, as it requires a deeper integration with the system, such as:

extensions to the relational algebra, and to query planning and optimization; automatic fine-tuning of the solver package; new fault tolerance mechanisms to deal with brittle solvers that fail due to unpredictable memory usage; support for nested queries, including complex joins (see Section 5.1.6) before or after package-level constraints. Also, in general, VG functions are parameterized by input tables which themselves (since data is usually stored in normalized form) are often the result of relational operations on base parameter tables. Combining relational operations with package queries requires the capabilities of a full-blown MCDB [80]. If the parameter tables themselves can be stochastic, then enhanced functionality of a system like SimSQL [29]–a database system for stochastic analytics–is needed. In SimSQL, the basic relational operators were engineered to deeply support Monte Carlo operations over relational data.

5.1.6 Handling joins

In this dissertation, we assumed that, in the presence of joins, the system simply evaluates and materializes the join result before applying the package-specific transformations. However, the materialization of the join result is not always necessary: **Direct** generates variables through a single sequential scan of the join result, and thus the join tuples can be pipelined into the ILP generation without being materialized. However, not materializing the join results means that some of the join tuples will need to be recomputed to populate the solution package. Therefore, there is a space-time trade-off in the consideration of materializing the join. Further, this trade-off can be improved with hybrid, system-level solutions, such as storing the record IDs of joining tuples to enable faster access during package generation.

5.1.7 Top-k package queries

In this thesis, we focused on producing the single optimal result for a package query with an optimization objective. Our algorithms, **Direct** and **SketchRefine**, are not designed to efficiently produce top-k packages, as ILP solvers typically return one solution. A naïve way of producing top-k results is to return one result at a time, and modify the query in each iteration, so as to exclude the previous result. However, such an approach is inefficient. Efficient top-k packages is an important and interesting research direction, which may benefit from solver-specific solutions.

5.1.8 Differentially private package queries

In this thesis, we assumed that both the input data and the output of package queries are managed by trustworthy entities. In real-life settings, the data is often handled by third-party software that cannot be fully trusted, and result packages should protect the identity of individuals and other entities. In differential privacy [81], one seeks to publicly release the results of a statistical aggregation query, such as counting the number of cases a certain drug causes cancer, without giving away information about the individuals who took part in the statistic, i.e., whether a certain person has cancer or not. Accurate differential privacy is hard to achieve even for simple SQL aggregation queries. Computing the result of a package query on differentially private data poses new challenges. If packages are computed over data that has been perturbed for privacy reasons, it might be hard to identify feasible packages and the quality of the returned packages might suffer. Solutions able to return high-quality packages might require a combination of techniques from robust optimization and differential privacy.

5.2 PQBE: Package queries by example

A well-specified package query [23, 24] will return the optimal collective decision that can possibly be made with the existing data: the best meal plan, the perfect choice of stocks to sell and buy, the perfect team to win the match, the ideal vacation plan to Hawaii. However, specifying the query itself is far from being easy for the majority of the users. This is true of standard SQL queries [77, 78]. It is also true of PaQL because package query are more expressive than SQL queries [25]. The source of difficulty can be identified in four key aspects:

Incomplete specification: Users may not to be able to exactly list all the fine-grained characteristics of what they need in a package. For example, they may remember to specify that they have a budget, but they may forget to express that they do not want to save money on food quality.

Incorrect specification: Users may not be able to correctly formalize their needs in the form of constraints. In fact, expressing complex constraints in a package query can sometimes become as hard as programming itself.

Infeasible or overly selective specification: Because they do not have exact knowledge of the data, users may ask for something unachievable or too restrictive. For instance, they may prefer single-leg flights, which may eliminate very cheap alternatives involving just two legs from the solution space.

Overly permissive specification: For a similar reason, they may express constraints that are too permissive.

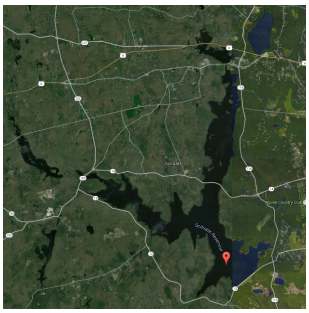
Although query specification can be hard, users are usually able to describe their *ideal package* by means of examples. An example package depicts the result that the user is expecting from the system in response to an information need that they have in mind.

The package example replaces the package query in this interaction: users need not to formalize their needs in PaQL, but, instead, they provide the system with an example of what they are looking for. This *query-by-example* paradigm has been successfully applied to standard database queries [5], but it has never been applied to results of combinatorial optimization problems such as packages.

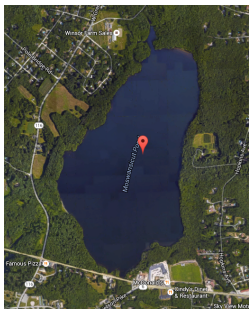
This interaction is very close to the typical interaction a user has with a *search engine*, and it has been extensively studied in *information retrieval* (IR) [11]. Creating an analogy between the *vector space model* [135] and package queries, documents in a vector space are the packages that the user is interested in finding. A *search query*, in this model, lies in the same vector space as the documents: it is effectively interpreted as an *example of a relevant document*: the model gives higher scores to documents that are more similar to the query in the vector space. or the *cosine kernel*, that is, the normalized dot product between the query and the document vectors.

Example 9 (Water Bodies)**.** A user is interested in identifying *water bodies*, such as *lakes*, *swamps*, *ponds*, *glaciers*, and *reservoirs* (see Figure 5.1 for examples). The user does not know how to "describe" water bodies in a query language such as PaQL. However, she knows what a water body should look like, and can thus provide the system with an *example water body*. The tuples in this example package only have color and transparency components *red*, *green*, *blue*, and *alpha*, and no location coordinates. This is because the user wants to describe the color attributes of a water body (for instance, having a lot of blue), but she does not want to restrict the results to any particular location.

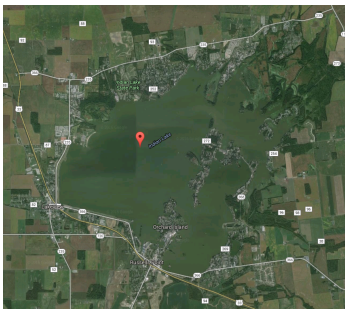
The system's objective is to identify *actual* water bodies, as packages, from the table U, based on the example provided by the user. In this work, we devise two new techniques for querying packages by example, each inspired by work done in the IR and DB areas,



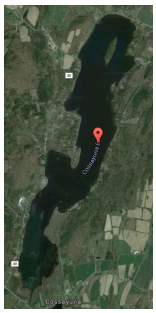
Scituate Reservoir



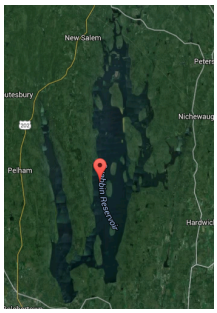
Moswansicut Pond



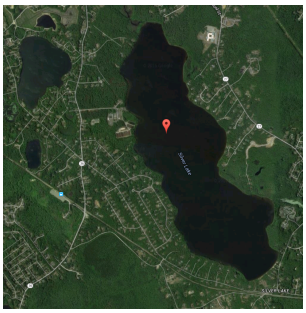
Indian Lake



Cossayuna Lake



Quabbin Reservoir



Silver Lake

Figure 5.1: Images of 6 of the water bodies used as ground truth in the experiments.

respectively. Users provide the system with an example package. The system's objective is to construct an actual package from the underlining data that answers the user's implied query as optimally as possible. In this section, we initiate the study of the PQBE problem by considering two potential techniques, reporting some preliminary results, and indicating future directions for research. The techniques are:

VCLUST**: Clustering in the vector space** This method first retrieves tuples from the input data that are as similar as possible to the tuples from the example package. It then clusters them according to some of the data dimensions to form the final packages (Section 5.2.1.1).

QSynth**: PaQL query synthesis** This method synthesizes a PaQL query based on some global properties of the example package. It then solves the PaQL query to produce results (Section 5.2.1.2).

We run experiments on a real-world dataset of US water bodies extracted from the US Geological Survey (USGS) [156]. Our experiments show that: VClust is able to identify many of the correct water bodies, usually favoring precision over recall; QSynth is also able to identify correct water bodies, favoring recall over precision, but at a much higher cost in running time. These initial experiments indicate that there is clearly room for improvement, motivating further research.

We consider the following slightly simplified definition of packages. Let us consider a relation R, $|R| = n$, with schema $R(a_1, \ldots, a_k)$, where $a_j \in \mathbb{R}$, $1 \leq j \leq k$, is the j-th attribute of R.

Definition 12 (Package)**.** *A* package P *is a subset of* R*, with the same schema as* R*. That is,* $P \subseteq R$*,* $P(a_1, \ldots, a_k)$*.*

Definition 13 (Example Package)**.** *An* example package $\tilde{P}$ *is a relation having as attributes a subset of the attributes of* R*, and being itself not necessarily a subset of* R*. That is,* $\tilde{P}(a_{i1}, \ldots, a_{ik'})$*, where* $(a_{i1}, \ldots, a_{ik'}) \subseteq (a_1, \ldots, a_k)$*, and possibly* $\tilde{P} \not\subseteq R$*.*

An algorithm for querying packages by example takes an example package $\tilde{P}$ as input and returns one or more packages $P_1, \ldots, P_m$. The returned packages should be relevant for the user's intent, as inferred on the basis of $\tilde{P}$. Each algorithm makes a different assumption regarding this notion of relevance, and uses different techniques for constructing the final packages.

5.2.1 Methods for querying packages by example

We now describe the two methods we devised for querying packages by example.

5.2.1.1 VClust: Clustering in a vector space

The first method interprets the example tuples from the example package $\tilde{P}$ as a set of *query vectors* in a vector space. The result packages $P_1, \ldots, P_m$ are constructed by: (1) retrieving the set of tuples from R that are as "similar" as possible to the example tuples, based on a similarity measure in the vector space; (2) clustering the retrieved tuples based on some of the dataset attributes. We now describe this method in greater detail.

Vector space embedding. Initially, we embed all of the tuples from R into a vector space $V(R)$. This embedding is straightforward as each tuple $t = (t.a_1, \ldots, t.a_k)$ from R is already a vector of numeric values $\langle t.a_1, \ldots, t.a_k \rangle$. The vectors in $V(R)$ are then trained and indexed using a particular model, such as TFIDF [143, 11] or LSI [95]. This step produces a new set of vectors, where each vector is trained on the entire set of input tuples from R.

Relevant tuples retrieval. Each example tuple from $\tilde{P}$ is then embedded into the same vector space $V(R)$. This produces a vector for each of the example tuples. These are treated as query vectors in $V(R)$. They are either averaged into a single query vector, or treated separately. These query vectors are then used to retrieve a certain number (e.g. 1000) vectors from $V(R)$, using a similarity measure in the vector space, such as the *cosine similarity* [143]. Each of the resulting vectors corresponds to a unique tuple $t \in R$. Thanks to the vector space model, tuples that resemble the example tuples are ranked higher.

If more than one ranking is produced at this step (because there were more than one query vector), the resulting rankings are aggregated using a rank aggregation technique such as CombMNZ [107, 140]. According to CombMNZ, if $s_i(t)$ is the score of tuple t in the i-th ranking, and $n(t)$ is the number of rankings that contain tuple t, the combined score of tuple t is:

$$s(t) = n(t) \sum_i s_i(t)$$

Package construction via clustering. The ranked tuples extracted from R are clustered into m clusters, based on some of the original attributes. Each final cluster constitutes one of the m resulting packages $P_1, \ldots, P_m$.

Package ranking. Finally, we rank the resulting packages based on two measures: (1) the sum of all the tuple-level scores of the tuples in the package, discounted by (2) how much the size of the result package differs from the size of the example package. The final score of a result package P is:

$$s(P) = \sum_{t \in P} s(t) - \left| |P| - |\tilde{P}| \right|$$

5.2.1.2 QSynth: synthesizing and solving a PaQL query

The second method tries to infer an underlying PaQL query from the example package $\tilde{P}$. It then solves this PaQL query to produce a single result package P.

Synthesizing a PaQL query. The algorithm first identifies a set of *constraint aggregates* $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_l\}$, among a finite set of possible aggregates. Typically, the set of possible aggregates includes: $COUNT(*)$ (the cardinality), $SUM(a_j)$, (the sum over attribute a_j),

and $AVG(a_j)$ (the average over attribute a_j), $1 \leq j \leq k$. A special *objective aggregate* $\mathcal{A}_{obj}$ from the same set of possible aggregates is also identified.

The algorithm then computes the set of *aggregate values* $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_l\}$ of $\tilde{P}$ by evaluating each of the constraint aggregates over $\tilde{P}$. Finally, given a *flexibility* f, $0 \leq f \leq 1$, the following PaQL query $\mathcal{Q}(\tilde{P})$ is constructed:

$\mathcal{Q}(\tilde{P})$: SELECT PACKAGE(R) AS P
FROM R
SUCH THAT $\mathcal{A}_1$(P) BETWEEN $\mathcal{V}_1(1-f)$ AND $\mathcal{V}_1(1+f)$ AND
...
$\mathcal{A}_l$(P) BETWEEN $\mathcal{V}_l(1-f)$ AND $\mathcal{V}_l(1+f)$
MAXIMIZE $\mathcal{A}_{obj}$(P)

Solving the PaQL query

To solve the synthesized package query we use either of our two methods **Direct** or **SketchRefine** (see Chapter 3). We briefly recall the main ideas of the two methods.

Direct first formulates the PaQL query $\mathcal{Q}(\tilde{P})$ as an *integer linear program* (ILP), using a set of transformation rules that preserve the feasibility and optimality of the query. It then employs an off-the-shelf ILP solver, as a black box, to get a solution to the ILP problem. Finally, it converts the ILP solution to the result package, and outputs the package.

The ILP problem that corresponds to query $\mathcal{Q}(\tilde{P})$ has a set of n, $n = |R|$, binary variables $x_1, \ldots, x_n$, one for each input tuple in R. Let t_i be the i-th tuple from R, x_i its corresponding binary variable, and $t_i.\mathsf{a}_j$ the value of attribute a_j of tuple t_i. As an example, consider the following package query:

$\mathcal{Q}_1$: SELECT PACKAGE(R) AS P

FROM R

SUCH THAT SUM(a_1) BETWEEN $\mathcal{V}_1(1-f)$ AND $\mathcal{V}_1(1+f)$ AND

...

SUM(a_l) BETWEEN $\mathcal{V}_l(1-f)$ AND $\mathcal{V}_l(1+f)$

MAXIMIZE COUNT($*$)

The ILP problem for $\mathcal{Q}_1$ is:

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^{n} x_i & \\ \text{s.t} \quad & \textstyle\sum_{i=1}^{n} t_i.\mathsf{a}_j \cdot x_i \geq \mathcal{V}_j(1-f) & \forall j = 1,\ldots,k \\ & \textstyle\sum_{i=1}^{n} t_i.\mathsf{a}_j \cdot x_i \leq \mathcal{V}_j(1+f) & \forall j = 1,\ldots,k \\ & x_i \in \{0,1\} & \forall i = 1,\ldots,n \end{aligned}$$

The **SketchRefine** method, instead of solving the entire ILP problem directly (which can be very inefficient), first breaks the problem down into multiple subproblems, and then uses the solver to solve the subproblems. The subproblems are generated based on a partitioning (computed offline, not a query time) of the dataset along all of the attribute dimensions. The subproblems are generated in such a way that solutions to previous subproblems are incorporated in subsequent problems, incrementally. This algorithm is an approximation to **Direct**, offering much faster computation.

5.2.2 Preliminary results

In this section, we report the results of the preliminary experiments that we conducted to study our methods for querying packages by example.

Dataset

In this work, we constructed an experimental dataset and used it to cast our problem onto real-world data. The dataset consists of a table U of raster satellite images of the US, with schema $U(red, green, blue, alpha, x, y)$, where each tuple corresponds to a unique pixel having color components *red*, *green* and *blue*, a transparency component *alpha*, and location coordinates (x, y) that correspond to geographic coordinates *longitude* and *latitude*.

We constructed the dataset using the US Geological Survey (USGS) [156]. We first extracted a ground-truth dataset of actual water bodies from the US, and loaded them into a PostgreSQL database extended with PostGIS [123], a library to support spatial objects and queries in PostgreSQL. We used QGIS [126] to load the data into the database. The resulting table G contained geometries (namely, multi-polygons), coordinates, and other meta-data for each water body.

Secondly, we retrieved raster images of the US, in .jp2 format, from the National Map Viewer of USGS.[1] We selected an area around Amherst, MA, that spanned Massachusetts and a few of the adjacent states. Each .jp2 raster file contained color components of pixels together with their geograpic coordinates. Although we restricted the data to an area around Amherst, this data was still too big to run our experiments. We selected a subset of the .jp2 files, some of which also contained water bodies, for a total of 11 water bodies. We loaded this data into PostgreSQL using a tool called `raster2pgsql` from the GDAL library [62], by aggregating groups of 100x100 pixels into tiles and averaging their color components and coordinates. The resulting table contained 133,000 tiles. We will refer to these tiles simply as pixels in the rest of the chapter.

[1] http://viewer.nationalmap.gov/viewer/

To generate our input table U, we also transformed the geographic coordinates (typically expressed in the EPSG:3785 coordinate system) into integer grid coordinates (x, y), so that the westernmost-southernmost point was mapped into $(0, 0)$, and the easternmost-northernmost point to $(4939, 3301)$. Because we selected a subset of all the raster image files, this grid contains some gaps.

In the final step of the dataset curation process, we constructed the true packages for each true water body by joining the two datasets U (of pixels) and G (the PostGIS table of water bodies) when regions in G overlapped with tuples in U. This resulted in a table W of 11 true water-body packages. In Section 5.1, we show some of these water bodies using images extracted from Google Maps.[2]

Querying water bodies by example

We run our experiments on a problem similar to Example 9. Here, we describe in detail how VCLUST and QSYNTH are used for the water bodies dataset.

VCLUST. To address this problem using the method described in Section 5.2.1.1, we proceed as follows:

1. We generate and train the vector space $V(U)$ using the color and transparency components *red*, *green*, *blue*, and *alpha* from the schema of the example water body. The coordinate components are not used during this training phase because not present in the example package.

2. We embed the example tuples in the same vector space $V(U)$. This generates a set of query vectors.

[2] `http://maps.google.com/`

3. We retrieve 1/10-th of the tuples from U, via $V(U)$, using the query vectors. This returns a ranking of tuples sorted by their likelihood of being "water body tuples".

4. We cluster these tuples based on their geographic coordinates x and y. Tuples belonging to the same cluster will form a single result package.

We employ DBSCAN [56] for clustering, as implemented by Scikit-learn [116], with the following parameters: Euclidean distance as the metric, 5 as the minimum number of samples in a neighborhood of a core point, 1.0 as the the maximum distance between two samples for them to be considered as in the same neighborhood, the Ball tree [91] as the data structure to compute nearest-neighbors. DBSCAN does not make any assumption on the number of clusters to generate. This is important because we do not know a priori the number of packages to produce. Other clustering algorithms, such as *k-means* [72], instead, only work with the number of clusters provided as input.

We experiment with two methods for generating the top tuples (step 3). The first method, called RankAggr, runs a query over the vector space model for each tuple in the example package, and then aggregates the results using CombMNZ [107, 140]. The second method, called QueryAvg, runs one single query on the vector space model, namely, the average of all the query vectors.

In our experiments, we use a simplified TFIDF weighting scheme. Specifically, consider the traditional IDF weight for color component c, defined as:

$$\mathrm{IDF}(c) = \log\left(\frac{n}{n(c)}\right)$$

where n is the size of the dataset and $n(c)$ is the number of pixels in the dataset having a non-zero component on color c. Using this scheme for the IDF weights is meaningless in our scenario because nearly every pixel has non-zero components in each of the colors

(including the transparency dimension as well), producing $\text{IDF}(c) = 0$ for each color c. Setting all IDF's to a constant (e.g. 1) solves the problem, but it does not account for example packages lacking some color components in all or some of the pixels. Therefore, we utilized a smoothed IDF measure defined as:

$$\text{IDF}(c) = \log\left(1 + \frac{n}{n(c)}\right)$$

Term frequencies in traditional text retrieval settings are normalized per document. This means that the TF weight for a particular pixel p and color component c would be:

$$\text{TF}(p, c) = \frac{value(c, p)}{\max\{value(c', p) : \forall c'\}}$$

where $value(c, p)$ is the value of color c of pixel p. We did some initial experiments with this scheme and found out that the retrieval model was performing very badly. A better normalization scheme, which we have used in our experiments, produces TF weights in $[0, 1]$ by dividing the color values by the overall maximum value that a color component can take on, that is, 255 for RGB data:

$$\text{TF}(p, c) = \frac{value(c, p)}{255}$$

QSynth. Following the method description presented in Section 5.2.1.2, QSynth constructs global constraints and objective criteria that capture water bodies. We experiment with two types of constraint aggregates. The first type is based on summations of the color and transparency components, *sum-aggregates*: SUM(red), SUM($green$), SUM($blue$), SUM($alpha$). The second type, instead, uses averages, *avg-aggregates*: AVG(red), AVG($green$), AVG($blue$), AVG($alpha$).

Furthermore, we add constraints to capture the *contiguity* of tuples that form a water body. We experiment with two types of contiguity: *strip-contiguity* and *box-contiguity*. Both types of constraints group tuples in a package by x (or y) coordinates, and make sure that the tuples in the corresponding y (or x) range is fully (or partially) covered.

The strip-contiguity is the loosest of the two. It ensures the contiguity of the pixels in a result package per *strip*. Given a contiguity requirement $\mathcal{C}$, $0 \leq \mathcal{C} \leq 1$, it can be expressed in PaQL as:

$$\begin{aligned}
&\mathcal{C} \leq \text{ALL } (\text{SELECT COUNT}(*)/(\text{MAX}(y) - \text{MIN}(y) + 1) \\
&\qquad \text{FROM P GROUP BY } x) \text{ AND} \\
&\mathcal{C} \leq \text{ALL } (\text{SELECT COUNT}(*)/(\text{MAX}(x) - \text{MIN}(x) + 1) \\
&\qquad \text{FROM P GROUP BY } y)
\end{aligned}$$

The box-contiguity requires that the result package be a rectangle of contiguous pixels (on the x and y coordinates). Non-contiguous pixels are allowed with a contiguity requirement $\mathcal{C}$. This constraint can be expressed in PaQL as:

$$\begin{aligned}
&\mathcal{C} \cdot (\text{SELECT MAX}(y) - \text{MIN}(y) + 1 \text{ FROM P}) \leq \text{ALL } (\\
&\quad \text{SELECT COUNT}(*) \text{ FROM P GROUP BY } x) \text{ AND} \\
&\mathcal{C} \cdot (\text{SELECT MAX}(x) - \text{MIN}(x) + 1 \text{ FROM P}) \leq \text{ALL } (\\
&\quad \text{SELECT COUNT}(*) \text{ FROM P GROUP BY } y)
\end{aligned}$$

Both strip-contiguity and box-contiguity can be linearized using known linearization tricks [19]. We report the full linearization of both types of constraints in Section 5.2.3. These types of linearization have the drawback of adding a substantial number of new variables and linear constraints to the original ILP problem, sometimes making the problem much more complex to optimize.

Finally, among all sets of tuples satisfying the color and contiguity constraints, we prefer the largest set. We can express this in PaQL with: MAXIMIZE COUNT(∗).

As an example, suppose $\tilde{P}$ is the following package:

red	$green$	$blue$	$alpha$
112	10	250	255
9	84	241	151
46	191	199	200

The resulting PaQL query, using avg-aggregates with $f = 0.2$ and box-contiguity with $\mathcal{C} = 0.9$ is:

```
SELECT      PACKAGE(R) AS P
FROM        R
SUCH THAT   AVG(red) BETWEEN 167*0.8 AND 167*1.2 AND
            AVG(green) BETWEEN 285*0.8 AND 285*1.2 AND
            AVG(blue) BETWEEN 690*0.8 AND 690*1.2 AND
            AVG(alpha) BETWEEN 606*0.8 AND 606*1.2 AND
            0.9 * (SELECT MAX(y) - MIN(y)+1 FROM P) <= ALL (
              SELECT COUNT(*) FROM P GROUP BY x) AND
            0.9 * (SELECT MAX(x) - MIN(x)+1 FROM P) <= ALL (
              SELECT COUNT(*) FROM P GROUP BY y)
MAXIMIZE    COUNT(*)
```

Experimental setup

In our experiments, we simulate a user providing an example water body in two ways: (1) by manually creating an example water body; (2) by using one of the true water bodies from table W.

We evaluate both the effectiveness and efficiency of the results produced by the methods. The effectiveness is evaluated against the ground truth of water bodies. We say that a true water body has been "hit", or "retrieved" by a package result if the package result covers its surface area by at least 10%, with at least 10% precision. More formally, given a package result P and a true water body T, we define the *package-level recall* and *package-level precision* of P against T as:

$$\text{PT-}recall(P,T) = |P \cap T|/|T|$$

$$\text{PT-}precision(P,T) = |P \cap T|/|P|$$

A single package result P can potentially include pixels from more than one true water body. In this case, the resulting PT-*recall*'s would increase, but the PT-*precision*'s would decrease. To decide whether P has hit a true water body, we average all the non-zero PT-*recall*'s and PT-*precision*'s against all true water bodies, and check whether they are both at least 0.10.

Once we know whether a single result P is a hit or not, we can compute the traditional *precision* and *recall* of the result packages $P_1, \ldots, P_m$ as:

$$recall = \frac{\text{\# of hits}}{11}$$

$$precision = \frac{\text{\# of hits}}{m}$$

To evaluate our results, we compute the *average precision* of the ranked packages $P_1, \ldots, P_m$ returned by the method. The average precision is defined as:

$$AP = \frac{\sum_{k=1}^{m} P(k) \cdot hit(k)}{11}$$

			Min	Avg	Max	Min	Avg	Max	
Query method	# Results	AP	PT-*recall*			PT-*precision*			Time (s)
RankAggr	100	0.038	0.021	0.115	0.191	0.082	0.816	1.000	0.01
QueryAvg	913	0.020	0.002	0.031	0.238	0.004	0.928	1.000	0.01

Table 5.1: Results of VClust, using the blue box as an example package.

where $P(k)$ is the *precision-at-k*, that is, the precision over packages P_1 through P_k only, and $hit(k)$ is 1 if the k-th result is a hit, and 0 otherwise. We also report the *mean average precision* (MAP) as the average of AP across multiple example packages.

Results and discussion

Performance against manually-constructed example. In this experiment, we construct a manual example of a water body. The manual example is a box of 100x100 identical blue pixels, that is, all having RGB components $(0, 0, 255)$, with no transparency and no location components.

Table 5.1 shows the results of using both RankAggr and QueryAvg. The second column show the number of packages returned by VClust using either of the two methods. The remaining columns report the evaluation results: average precision (AP), and minimun, average and maximum PT-*recall* and PT-*precision*. The last column reports the average running time of producing a single package result.

The average precision, in both cases, is very low. A motivation for it can be found in the average and maximum PT-*recall*, which is close to 0.10 (recall that a hit can occur with PT-*recall* ≥ 0.10).

These results show that VClust is not very effective in identifying water bodies when presented with the blue-box example package. The method is, however, extremely fast in

					Min	Avg	Max	Min	Avg	Max	
$\check{P}$ No.	$\check{P}$ Name	$\check{P}$ Size	# Results	AP	PT-*recall*			PT-*precision*			Time (s)
1	Cossayuna Lake	18	442	0.022	0.004	0.037	0.191	0.001	0.913	1.000	0.01
2		246	770	0.006	0.002	0.030	0.381	0.004	0.875	1.000	0.01
3	Scituate Reservoir	270	734	**0.606**	0.004	0.033	0.381	0.003	0.890	1.000	0.01
4		146	1009	0.008	0.002	0.032	0.429	0.004	0.839	1.000	0.01
5	Moswansicut Pond	241	774	**0.370**	0.004	0.028	0.381	0.002	0.833	1.000	0.01
6	Indian Lake	236	804	0.020	0.004	0.030	0.238	0.003	0.890	1.000	0.01
7		261	691	<u>0.000</u>	0.004	0.027	0.046	0.001	0.761	1.000	0.01
8		85	954	0.030	0.002	0.037	0.476	0.004	0.800	1.000	0.01
9	Silver Lake	475	994	0.001	0.002	0.027	0.381	0.003	0.848	1.000	0.01
10	Quabbin Reservoir	241	795	0.028	0.002	0.026	0.190	0.006	0.895	1.000	0.01
11		21	872	0.007	0.002	0.033	0.381	0.006	0.909	1.000	0.01

Table 5.2: Performance of VCLUST with RANKAGGR, using true water bodies as example packages. The best and worst average precisions are in bold and underlined, respectively. The MAP of this experiment is 0.100**.**

producing package results. In the following experiment, we use better package examples and show that the performance of VCLUST improves with more realistic examples.

Performance against true examples. In this experiment, we use each one of the true water bodies as example packages.

Results of VCLUST. We first report the results of VCLUST that uses RANKAGGR for generating the tuple-level ranking. The results of this experiment are reported in Table 5.2. Of the 11 true water bodies used as example packages, the two that were able to retrieve true water bodies with high average precision were the Scituate Reservoir and the Moswansicut Pond. However, most of the time, the average precision is low, with a case (package 7) where none of the 691 package results hit a true water body. The average running time is always low due to the inverted index used to retrieve the relevant tuples and the fast DBSCAN clustering algorithm.

The PT-*recall* and PT-*precision* values provide more insight into these results. The average PT-*recall* is always below 0.10, which motivates the poor performance on average precision (recall that a hit can occur with PT-*recall* ≥ 0.10). The maximum PT-*recall*

					Min	Avg	Max	Min	Avg	Max	
$\check{P}$ No.	$\check{P}$ Name	$\check{P}$ Size	# Results	AP	PT-*recall*			PT-*precision*			Time (s)
1	Cossayuna Lake	18	317	<u>0.000</u>	0.004	0.033	0.085	0.001	0.378	0.909	0.01
2		246	761	0.007	0.004	0.040	0.381	0.003	0.818	1.000	0.01
3	Scituate Reservoir	270	817	0.039	0.004	0.035	0.381	0.003	0.875	1.000	0.01
4		146	794	0.006	0.004	0.041	0.381	0.003	0.824	1.000	0.01
5	Moswansicut Pond	241	800	0.039	0.004	0.037	0.381	0.003	0.854	1.000	0.01
6	Indian Lake	236	856	0.030	0.004	0.037	0.381	0.003	0.871	1.000	0.01
7		261	516	<u>0.000</u>	0.004	0.029	0.059	0.002	0.848	1.000	0.01
8		85	751	0.008	0.002	0.037	0.381	0.003	0.825	1.000	0.01
9	Silver Lake	475	1112	0.039	0.002	0.033	0.381	0.003	0.884	1.000	0.01
10	Quabbin Reservoir	241	1171	**0.050**	0.002	0.033	0.381	0.006	0.894	1.000	0.01
11		21	1109	0.046	0.002	0.033	0.381	0.004	0.881	1.000	0.01

Table 5.3: Performance of VCLUST **with** QUERYAVG**, using true water bodies as example packages. The best and worst average precisions are in bold and underlined, respectively. The MAP of this experiment is** 0.024**.**

on package 7 is below 0.10, which explains why its average precision is 0. Interestingly, the maximum PT-*precision*'s are all 1, and all of the average PT-*precision*'s are very high, usually above 0.70 or 0.80. This means that: (1) the example water body was hit, and (2) the method hit other water bodies with very high precision. This strong difference in performance between PT-*recall* and PT-*precision* means that the algorithm is not able to cover a great portion of a true water body, but when it does so, it does not make much mistakes. This could be a result of the clustering algorithm being used: the clustering algorithm may have broken down pieces of the same water body and reported them as different packages. In this case, the bad performance on average precision could be mitigated by using a more sophisticated clustering algorithm.

In Table 5.3, we report the results of VCLUST that uses QUERYAVG to generate the tuple-level ranking. QUERYAVG performs generally worse than RANKAGGR, as demonstrated by the worst MAP (0.024 against 0.100 of RANKAGGR). Similarly to RANKAGGR, the true water body 7 is a bad example as it is not able find other true lakes. However, with QUERYAVG, this bad performance happens also with true package 1. In some cases, on the

$\tilde{P}$ No.	$\tilde{P}$ Name	$\tilde{P}$ Size	PT-*recall*	PT-*precision*	Time (s)
1	Cossayuna Lake	18	<u>0.000</u>	<u>0.000</u>	263.37
2		246	0.542	0.118	1974.41
3	Scituate Reservoir	270	0.510	0.227	551.07
4		146	0.316	0.120	291.68
5	Moswansicut Pond	241	0.490	0.216	558.68
6	Indian Lake	236	0.456	0.214	423.56
7		261	0.503	0.162	589.69
8		85	0.154	0.103	425.43
9	Silver Lake	475	0.521	0.123	663.23
10	Quabbin Reservoir	241	**0.631**	**0.324**	315.01
11		21	<u>0.000</u>	<u>0.000</u>	244.27

Table 5.4: Performance of QSynth with sum-aggregates and strip-contiguity, using true water bodies as example packages. The best and worst results are in bold and underlined, respectively.

contrary, QueryAvg performs better than RankAggr (as in Silver Lake and Quabbin Reservoir), but the improvement is not substantial.

Results of QSynth. We present the results of QSynth using sum-aggregats with $f = 0.5$ and strip-contiguity with $\mathcal{C} = 0.5$. The results of this experiment are reported in Table 5.4. Because QSynth only returns one single package result, we do not evaluate its performance using average precision (which could only reach 1/11 at best, in this case). Instead, we only report the PT-*recall* and PT-*precision* of the returned package, and the running time for generating it. For 9 of the 11 example packages (packages 2 through 10), the result package was a hit. The PT-*recall* of the packages returned by QSynth are usually substantially higher than the maximum PT-*recall* of VClust with RankAggr, albeit their PT-*recall* usually lower. This suggests that QSynth is able to identify bigger portions of water bodies than VClust, at the cost of precision. The result on Quabbin Reservoir is particularly interesting, as QSynth was able to return a true water body with both high precision and recall, whereas this particular lake performed quite badly with the VClust method. The running time of QSynth is aways substantial, due to the time

$\tilde{P}$ No.	$\tilde{P}$ Name	$\tilde{P}$ Size	PT-*recall*	PT-*precision*	Time (s)
1	Cossayuna Lake	18	0.004	0.003	2049.16
2		246	0.017	0.011	1995.23
3	Scituate Reservoir	270	<u>0.000</u>	<u>0.000</u>	1985.43
4		146	<u>0.000</u>	<u>0.000</u>	1988.21
5	Moswansicut Pond	241	0.167	**0.176**	1984.32
6	Indian Lake	236	<u>0.000</u>	<u>0.000</u>	1985.75
7		261	0.010	0.007	2039.68
8		85	<u>0.000</u>	<u>0.000</u>	1984.08
9	Silver Lake	475	0.085	0.033	1985.22
10	Quabbin Reservoir	241	**0.509**	0.046	1994.61
11		21	<u>0.000</u>	<u>0.000</u>	1987.35

Table 5.5: Performance of QSynth with avg-aggregates and strip-contiguity, using true water bodies as example packages. The best and worst results are in bold and underlined, respectively.

spent solving the ILP problem corresponding to the PaQL query. Most of this complexity is caused by the contiguity constraints.

We conclude with the results of QSynth using avg-aggregates instead of sum-aggregates, shown in Table 5.5. Using averages as aggregates performs much worse than using summations. Not only the overall effectiveness degrades, but also the running time increases by a factor of x4. This sharp increase in running time is caused by the fact that avg-aggregates allow many more packages to be feasible, as packages of different sizes can achieve the same average values on the color components.

Discussion

We presented different methods for querying packages by example. We performed experiments where packages represented water bodies from a dataset of raster images of the US, and showed that the two methods are able to identify water bodies from examples. The results presented in this work motivate further research. In particular, we think that the results of VClust can be improved with more sophisticated clustering methods. As

a next step, we would like to study methods for improving the answers by interacting with the user. Relevance feedback [133] could be used to improve the quality of the initial example package and improve the results of either the VCLUST or the QSYNTH method.

5.2.3 Linearization of contiguity constraints

Let us consider the following strip-contiguity constraint:

$\mathcal{C} \leq$ ALL (SELECT COUNT$(*)$/(MAX(y) − MIN$(y)+1$) FROM P GROUP BY x)

Knowing a priori all u, $u \geq 1$, the distinct values of x, denoted as $g_1(x), \ldots, g_u(x)$, this constraint can be re-interpreted as a set of u constraints, where the k-th constraint is of the form:

$\mathcal{C} \leq$ (SELECT COUNT$(*)$/(MAX(y) − MIN$(y)+1$) FROM P WHERE $x = g_k(x)$)

We now proceed by showing how the above constraint can be linearized.

Pre-processing step. Using a SQL query, we compute the MAX(y) and MIN(y) FROM R (the entire input dataset) WHERE $x = g_k(x)$.

Constants. Consider the following constants:

- $Y_{max} =$ MAX$(y) * 1$
- $Y_{min} =$ MIN$(y) * 0$
- $Y_{range} = Y_{max} - Y_{min} + 1$
- $Y_{diff} = Y_{max} - Y_{min}$

New variables. We add the following new variables to the ILP:

- New integer variable M_y, with bounds $Y_{min} \leq M_y \leq Y_{max}$, which encodes $\mathsf{MAX}(y)$ WHERE $x = g_k(x)$.
- New integer variable m_y, with bounds $Y_{min} \leq m_y \leq Y_{max}$, which encodes $\mathsf{MIN}(y)$ WHERE $x = g_k(x)$.
- New set of Y_{range} binary variables $M_1, \ldots, M_j, \ldots M_{Y_{range}}$ that "linearly expand" M_y. The meaning of one of these variables is $M_j = 1 \iff M_y = j$.
- New set of Y_{range} binary variables $m_1, \ldots, m_j, \ldots m_{Y_{range}}$ that "linearly expand" m_y. The meaning of one of these variables is $m_j = 1 \iff m_y = j$.

Target constraint. The contraint we would like to encode is the following:

$$\sum_i \mathbb{1}(t_i.x = g_k(x)) * x_i - \mathcal{C} * M_y + \mathcal{C} * m_y \geq \mathcal{C}$$

where $\mathbb{1}$ is the indicator function.

Binary expansion constraints. These constraints ensure that one and only one of the M_j and one of the m_j are set to 1:

$$\sum_j M_j = 1$$
$$\sum_j m_j = 1$$

MAX and MIN constraints. Using their binary expansions, for every tuple $t_i \in R$ with $t_i.x = g_k(x)$, let us $j = t_i.y - Y_{min}$, and add the following constraints:

MAX(y) WHERE $x = g_k(x)$:

$$t_i.y * x_i \leq M_y \leq t_i.y * x_i + Y_{diff} * (1 - M_j) + Y_{diff} * (1 - x_i)$$

MIN(y) WHERE $x = g_k(x)$:

$$t_i.y * x_i - Y_{diff} * (1 - m_j) \leq m_y \leq t_i.y * x_i + Y_{diff} * (1 - x_i)$$

The box-contiguity constraints can be linearly encoded following a similar approach.

5.2.4 SuDocu: Summarizing documents by examples using PaQL

In this section, we consider a specific application of PQBE–document summarization by example–and build initial solutions for it. Our interface, SuDocu, presented in this section, and our initial results show that applying the "by example" paradigm to document summarization has great potential for enhancing the ability of summarization systems to handle very subjective intents that cannot be otherwise easily expressed by users. We present initial results showing that building summaries as the result of a package query produces good results. However, further research is necessary to improve the quality of the summaries produced, comparing it against the summarization methods traditionally developed in the natural language processing literature for generic and query-based summarization, and to study the usability of our summarization by example via a user study.

Document collections, such as Wikipedia, contain a wealth of information that can assist in many tasks. Yet, finding the right information quickly and easily is still a big challenge, despite all the advances in search engine technology, natural language processing, and machine learning. Consider the following scenario:

Example 10 (Trip Planning)**.** Arnob wants to plan visits to interesting places around the USA. She wants to know interesting locations and typical weather conditions for each state, but finding this information on the Web for 50 states is tedious and time-consuming. She knows that Wikipedia contains all the information she needs, but each page is large and full of facts that are not relevant to her intent (e.g., demographics, law, etc.). Arnob can manually extract relevant summaries of at most 3 pages, by selecting a small set of sentences that correspond to her specific information needs (interesting places and weather). But to thoroughly research her options, she needs an automated way to do this for the remaining 47 states.

Surprisingly, today's technology cannot help Arnob! A search engine, like Google, is good at finding which web pages are likely to contain relevant information, but it would require many queries and Arnob would need to be very thoughtful about search keywords in order to collect the relevant information for all 50 states. Arnob tried to use Natural Language Processing (NLP) and Machine Learning (ML) techniques and found that *text summarization* tools may be helpful. However, most text summarization tools are "generic": they produce summaries that are not tailored for her personal preferences and specific information needs. The summaries she obtained from these tools did not cover all important aspects of her task, but rather provided general information about the state's politics, law, education, etc. Arnob found that some summarization tools can be tailored with a user intent, and require a natural language question to express it. She picked a question answering system, like Alexa, and issued the following question: "What are some interesting places in Massachusetts and how extreme is the weather there?" Unfortunately, the system could not understand what Arnob meant by "interesting places"—since interestingness is a very personal concept—and returned her sentences about places of general interest: MIT, Harvard Square, and Boston Library.

Arnob is interested in natural sites: parks, lakes, mountains, seas, etc. While particular preferences may be hard to express precisely with a query, it is easy for Arnob to identify relevant sentences within a document. For example, Arnob selected the following sentences from Utah's Wikipedia page as most relevant to her needs:

Example 11 (Personalized Summary of Utah)**.** The state of Utah relies heavily on income from tourists and travelers visiting the state's parks and ski resorts. Today, Utah State Parks manages 43 parks and several undeveloped areas totaling over 95,000 acres of land and more than 1,000,000 acres of water. With five national parks (Arches, Bryce Canyon, Canyonlands, Capitol Reef, and Zion), Utah has the third most national parks of any state after Alaska and California. Temperatures dropping below 0 hould be expected on occasion in most areas of the state most years.

She would like to extract something similar to the summary of Example 11 for each of the 50 states. Luckily, she can now use SuDocu, a personalized DOCUment SUmmarization system, that enables users to specify their summarization intent by a few *example summaries* and produces *personalized* summaries for new documents. SuDocu is an instance of a *query-by-example* system [58], tailored for text document summarization. The key motivation of SuDocu is that asking a user to provide examples of their desired answers, rather than vague questions, is a more effective way to learn the true intent, especially for a complex summarization intent involving multiple topics, e.g., interesting places *and* weather.

In this section, we introduce SuDocu [4], an end-to-end system that achieves *example-driven personalized* document summarization. The key idea is to view summarization as a combinatorial optimization problem where we want to extract an optimal set of sentences to form the summary, subject to the constraint that the summary's overall topic coverage should be *close* to that of the examples. We model topics of the documents using a

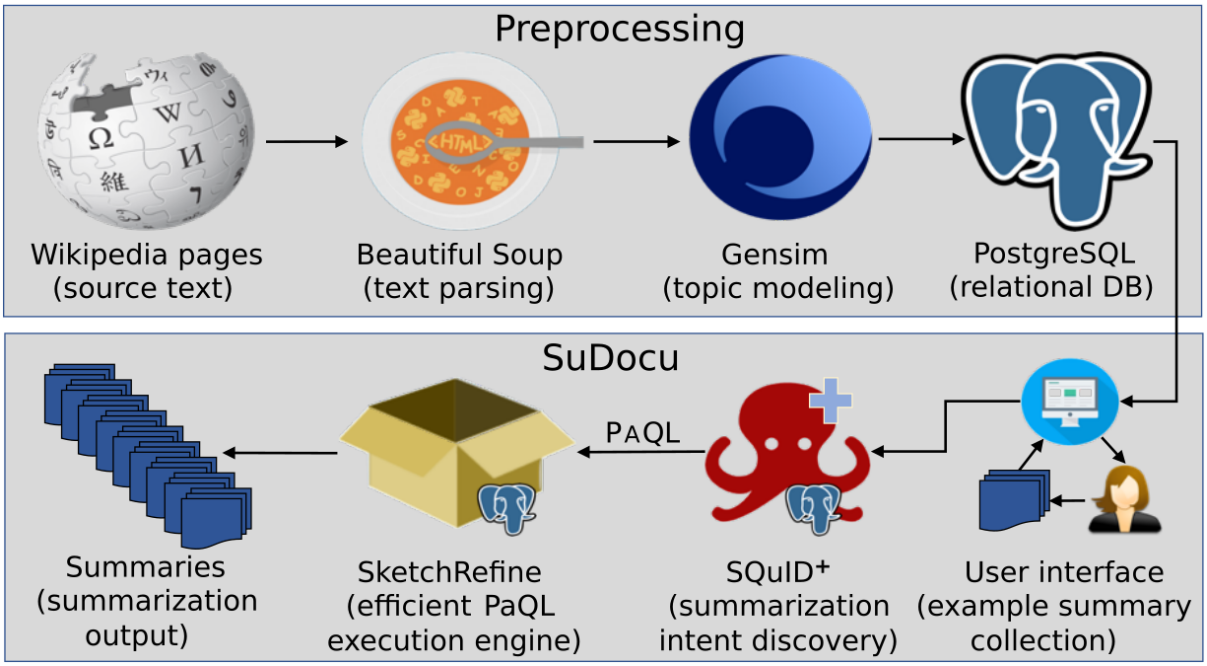


Figure 5.2: The SuDocu architecture. SuDocu combines SQuID$^+$ and SketchRefine in a novel way to summarize documents by example.

standard LDA approach [21], adapt our prior work for example-driven semantic similarity discovery [58] to create the constraints, and solve the resulting integer linear program using our techniques for scalable package queries (Chapter 3).

We show how a user like Arnob can use SuDocu for their Trip Planning task. The user observes first-hand how SuDocu detects their summarization intent from only a few example summaries of a few documents, and then efficiently produces summaries of new documents matching their intents. Users are free to specify their own intent by choosing different example summaries.

We now provide a solution sketch for SuDocu. Figure 5.2 depicts SuDocu's end-to-end pipeline. SuDocu pre-processes a corpus of documents by extracting all the sentences, automatically identifying all the topics, and assigning topic scores to each sentence. After preprocessing, the user can interact with SuDocu's interface and issue example summaries

Topic	Intuitive meaning	**Top related words and their associated weight** (ordered by decreasing weight)
1	*politics*	(governor, 0.015), (election, 0.013), (vote, 0.011), (democratic, 0.011), (majority, 0.009), (presidential, 0.008)
2	*legislature*	(century, 0.012), (passed, 0.011), (legislature, 0.010), (constitution, 0.009), (created, 0.007), (law, 0.006), (political, 0.006)
3	*urbanization*	(population, 0.077), (largest, 0.052), (city, 0.029), (percent, 0.019), (metropolitan, 0.012), (capital, 0.011), (people, 0.011)
4	*economy*	(major, 0.027), (economy, 0.018), (largest, 0.013), (industry, 0.013), (billion, 0.011), (production, 0.011), (oil, 0.009)
5	*demography*	(american, 0.029), (people, 0.021), (native, 0.018), (french, 0.015), (century, 0.015), (settlers, 0.012), (tribes, 0.010)
6	*climate*	(climate, 0.017), (feet, 0.011), (temperature, 0.010), (rail, 0.010), (forests, 0.009), (summer, 0.009), (winter, 0.009)
7	*location*	(north, 0.035), (west, 0.033), (south, 0.030), (east, 0.029), (southern, 0.022), (eastern, 0.020), (region, 0.020)
8	*taxes*	(tax, 0.056), (income, 0.030), (rate, 0.029), (ranked, 0.021), (nation, 0.021), (sales, 0.017), (average, 0.015), (capita, 0.014)
9	*education*	(government, 0.039), (school, 0.029), (county, 0.025), (public, 0.025), (federal, 0.023), (schools, 0.022), (law, 0.016)
10	*general*	(national, 0.007), (major, 0.006), (popular, 0.005), (system, 0.004), (founded, 0.004), (home, 0.004), (construction, 0.004)

Figure 5.3: Topics of Wiki pages of 50 states (extracted using topic modeling), their intuitive meaning, and top related words with associated weights.

to specify their intent. We first describe how we model the user intent, and then discuss preprocessing, summarization intent discovery, and summary generation.

Modeling personalized extractive summaries. Following prior work on text summarization [98], we model the personalized summarization as an optimization problem. Given the example summaries, we define the optimal summary as the one that maximizes a user-defined merit score (discussed later) such that the topic-coverage of the summary is similar to that of the example summaries. In SuDocu, we construct a linear constraint on topic-coverage for each topic, allowing scalable solution methods. We express the optimization problem as a package query.

Preprocessing. The first step of SuDocu involves extracting sentences from documents. In our implementation, we use Beautiful Soup, a library for extracting content from HTML pages. We then identify all of the topics in the extracted sentences.

We use the well-known *Latent Dirichlet Allocation (LDA)* topic model [21], in which a learned (latent) topic is represented as a set of weights assigned to the words in the vocabulary, and a sentence is viewed as a set of weights assigned to the topics. (Sentences here play the role of documents in [21].) The weight of a word (resp., topic) represents its relative importance to the topic (resp., sentence). For our implementation, we used Gensim, a standard NLP library that offers LDA-based topic modeling. Figure 5.3 shows the topics learned from the Wikipedia pages of 50 US states. In general, we can plug in any topic modeling technique into SuDocu.

The LDA topic weight of a sentence scores the relevance of a particular sentence to a particular topic. For example, the first sentence from the Example Summary 11, "The state of Utah relies heavily on income from tourists and travelers visiting the state's parks and ski resorts", would score high on "economy" and low on "education". Once sentences are encoded into the topic space, a sentence s (within document d) and its merit and topic-wise scores form a tuple of the form $\langle d, s, m\_score, s.T_1, s.T_2, \ldots, s.T_m \rangle$, where m_score is the merit score of s (see below) and $s.T_j$ denotes the score of s against topic T_j. We store these tuples into a PostgreSQL database.

Summarization intent discovery. To discover the summarization intent from example summaries, we extend the example-driven semantic similarity discovery approach of SQuID [58]; we call our extension SQuID$^+$. Whereas SQuID synthesizes SQL selection queries to retrieve tuples that are similar to user-specified example tuples, SQuID$^+$ synthesizes package queries to retrieve summaries (i.e., sets of tuples) that are similar to the user-specified example summaries. SQuID would treat a single sentence as an

example tuple; in contrast, SQuID$^+$ considers a *set* of sentences (summary) as an example package. Further, it aims to retrieve the summary with the highest utility (maximizing total m_score) among these similar summaries. To discover similarities among example summaries, we compute the topic-wise aggregate score for each example summary by summing the topic-wise scores of its sentences. That is, the score of example E_i against topic T_j is $E_i.T_j = \sum_{s \in E_i} s.T_j$. Now we specify the global topic-coverage predicate for T_j, given a set of examples $\{E_1, E_2, \dots\}$, as follows:

$$\text{SUM}(T_j) \text{ BETWEEN } \min_i E_i.T_j \text{ AND } \max_i E_i.T_j$$

Thus the aggregate score for each topic T_j in the summary must lie between the minimum and maximum aggregate scores in the examples; i.e., viewing each E_i as a point in topic space, the summary must lie within the bounding hyperrectangle of the examples.

One can further fine-tune the above constraint bounds: e.g., if most examples scored very high against a topic and only a few scored low, increase the minimum score threshold for that topic. In general, SuDocu can accept any package constraint derivation mechanism and is not limited to SQuID$^+$.

From the set of "feasible" summaries that satisfy the topic constraints, we want to select the "best" one. More precisely, we aggregate a per-sentence, user-defined "merit" score over the sentences in a summary to obtain the summary's merit score; we then seek the feasible summary having the highest merit score. Different definitions of merit are possible. If, e.g., the merit score of every sentence is -1, then maximizing the merit is equivalent to finding the shortest feasible summary. In our implementation, the merit score $m\_score(s)$ of a sentence $s = (w_1, \dots, w_J)$ comprising J words (with stop words excluded) is defined as $m\_score(s) = \sum_{j=1}^{J} F(w_j)$, where $F(w)$ is the normalized frequency of word w in the

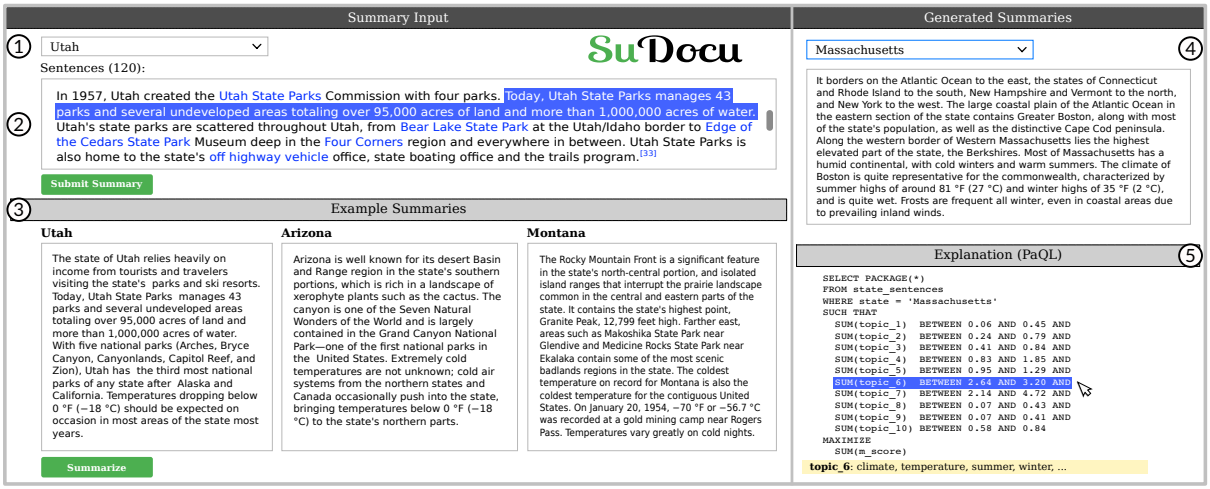


Figure 5.4: The SuDocu demo: ① the user selects a document for manual summarization, ② the user selects sentences from the document to construct an example summary, ③ the user views the example summaries, edits them if necessary, and submits them to request for summarization intent discovery, ④ the user specifies a new document to summarize and SuDocu produces a personalized summary of it, ⑤ PaQL query that captures the summarization intent.

corpus. Thus the more "important" (high corpus-frequency) words that a sentence contains, the higher its merit score.

The complete PaQL query is formulated as in Figure 5.4. Each tuple of the input relation corresponds to a sentence, and the attributes comprise the sentence and document IDs, along with the merit and topic-wise scores. The objective function to be maximized is the summary merit score SUM(m_score), and the WHERE clause ensures that only sentences from the document of interest are considered.

Efficient summary generation. Once the PaQL formulation of a package query is completed, the last step is to execute it. Package queries are combinatorial in nature, and solving them in general is NP-hard. If the problem is small enough, we can translate a package query directly into an equivalent integer linear program that can be solved with off-the-shelf

softwares. For each tuple t_i in the input relation, the translation assigns a binary decision variable x_i corresponding to the inclusion/exclusion of t_i in the answer package. When there are so many candidate sentences that the solver either cannot load the problem in main memory or fails to find a solution, we apply the **SketchRefine** algorithm [25], a divide-and-conquer approach that returns a near-optimal solution of the PaQL query having a provable approximation guarantee. SuDocu then presents the optimal set of sentences as the summary to the user, along with the PaQL query that encodes the summarization intent.

We build SuDocu on the Wikipedia pages of 50 US states to show how it can accurately detect the user's summarization intent and efficiently produce effective personalized summaries. We describe the user's interaction through five steps (Figure 5.4), first impersonating Arnob (a nature enthusiast) and then Bruno (an economics student). We annotate each step with a circle in Figure 5.4.

Impersonating Arnob. In our first scenario, the user impersonates Arnob of Example 10.

Step ① (Document selection for manual summarization): First, the user selects a state to manually summarize. Selecting a state displays all of the sentences from its Wikipedia page. In our screenshot, the user first selects Utah.

Step ② (Manual summarization): The user adds relevant sentences to the summary (by highlighting them) or removes previously selected sentences to refine the summary. Since Arnob is a nature enthusiast, the user picks sentences that mostly talk about parks, ski resorts, plants, canyons, etc. Moreover, since Arnob wants to know about the state's climate, the user also selects a few sentences about temperature. After summarizing Utah, the user repeats steps ① and ② for Arizona and Montana.

Step ③ (Summary submission): After manual summarization, the user can view the example summaries, editing them if needed. Once the user is satisfied, they request SuDocu

to discover their summarization intent. SuDocu processes the example summaries and generates a PaQL query that encodes this intent.

Step ④ (New document summarization): Since Arnob plans to visit the east coast, the user selects Massachusetts. SuDocu executes the PaQL query, adding state = 'Massachusetts' to the WHERE clause. SuDocu shows the returned package as the summary of Massachusetts. Massachusetts is by the Atlantic Ocean, and has no canyons or big mountains. Although the user never provided example sentences about oceans, SuDocu was still able to figure out that oceans would be among the most interesting places in Massachusetts based on topic similarity to canyons and parks. The summary also contains a few lines about temperature, cold winter, and warm summer, just as Arnob wants.

Step ⑤ (Summarization intent explanation): In the explanation panel, SuDocu shows the PaQL query—the underlying mechanism to produce new summaries. The user can edit the PaQL query directly to further refine their summarization intent. The query gives the insight that the user is mostly interested in topic_6 (climate) and topic_7 (location). Since the topic name is not clear from the query, the user hovers on topic_6, revealing the most related words for that topic: climate, temperature, summer, etc.

Impersonating Bruno. Bruno wants to write a report summarizing the economy of all 50 US states. Bruno is smart. So, instead of doing all the work by himself, he decides to use SuDocu. In our second scenario, the user impersonates Bruno. The steps are identical to the first scenario, but the summarization intent is completely different. The user now selects sentences that represent the state's economy. For example, for Utah, the user picks the sentence "The state has a highly diversified economy, with major sectors including transportation, education, ...". On completion of this task, Bruno gets a completely different summarization result than Arnob.

Generality. Besides the scenarios described so far, a user of SuDocu can also issue their own summarization intent using different example summaries. Further, they can also plug their own datasets into SuDocu.

Our interface showcases the ease and effectiveness of summarization by example. By formulating summarization intent as an optimization problem gleaned from a small set of user-provided examples, SuDocu can efficiently compute concise and informative personalized summaries. Further research is necessary to improve the quality of the produced summaries, as they sometimes include bad sentences that should be filtered out before package evaluation. The SuDocu interface would also benefit from a more user-friendly interaction mechanism for modifying the query constraints and objective, and a user study would be necessary to assess whether users like issuing subjective intents via example or with a more traditional querying mechanism.

5.3 IPE: Incremental package evaluation

At the core of our package evaluation methods, **Direct** is used as a black-box method to solve each subproblem. Treating the subproblem evaluation as a black box is a powerful abstraction: it allows our **SketchRefine** strategies to benefit from using alternative evaluation algorithms at its core, while the results of our theoretical analysis still hold (Section 3.3 in Chapter 3). In this section, we explore the potential of improving the performance of **Direct** directly, thus, slightly "lifting the lid" on this black box and exploiting some of its logic. Specifically, we will study the impact of *preconditioning*, i.e., an initial assignment of the variables, to the ILP solver's performance. The intuition is that providing the solver with a "good" starting package can reduce the search space and allow the solver to reach a solution faster.

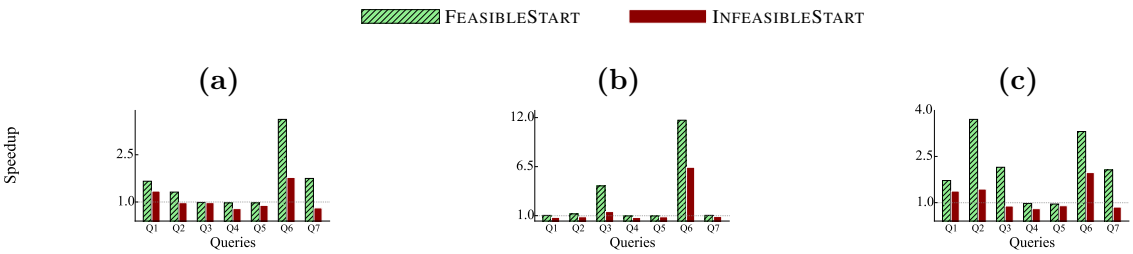


Figure 5.5: Average speedup provided by FeasibleStart (green bars) and InfeasibleStart (red bars) compared to NoStart, across different query sequences and different methodologies for sequence creation. (a) All constraints, fixed μ; (b) One constraint, fixed μ;(c) Random constraints, random μ. The results show that starting packages do not always improve the performance, but feasible starting packages generally offer better speedup than infeasible ones.

In this section, we do not present a particular method for identifying appropriate starting packages; our goal is to evaluate through a preliminary empirical analysis whether such a method can improve the efficiency of package evaluation in a meaningful way. Our analysis explores the following questions: (1) How does a starting package solution impact the runtime of **Direct**? (2) Does the feasibility of the starting package make a difference?

We evaluate the effect of seeding the solver with two types of starting solutions: packages that already satisfy the query's constraints (feasible), and packages that do not (infeasible). We use the Galaxy workload to construct sequences of queries with increasing strictness. Given a query $\mathcal{Q}$, we construct the sequence $(\mathcal{Q}_1, \ldots, \mathcal{Q}_r)$, such that $\mathcal{Q}_{i+1}$ has stricter constraints than $\mathcal{Q}_i$: if $\mathcal{Q}_i$ has a constraint $\mathsf{SUM}(\mathsf{attr}) \geq 2$ and the optimal solution to $\mathcal{Q}_i$ has a value 2.2 for this sum, then this constraint for $\mathcal{Q}_{i+1}$ becomes $\mathsf{SUM}(\mathsf{attr}) \geq 2.2 + \mu$, for a small constant $\mu > 0$. We construct three sequences for each query as follows:

1. we modify *all* constraints at every step of the sequence with a fixed μ,
2. we modify *only one* constraint at a time with a fixed μ,
3. we modify a random set of constraints with a random μ.

For all sequences $(\mathcal{Q}_1, \ldots, \mathcal{Q}_r)$, the solution for $\mathcal{Q}_i$ is feasible for $\mathcal{Q}_{i-1}$, but the solution for $\mathcal{Q}_{i-1}$ is not feasible for $\mathcal{Q}_i$. We construct sequences of length up to 20 for each of the 7 queries in the Galaxy workload. Sequences can have fewer than 20 queries if constraint changes cause a query to become infeasible. We execute each query $\mathcal{Q}_i$ in a sequence using **Direct** in three ways:

NoStart: Providing no starting solution.

FeasibleStart: Preconditioning with the optimal solution to $\mathcal{Q}_{i+1}$, which is a *feasible* package for $\mathcal{Q}_i$.

InfeasibleStart: Preconditioning with the optimal solution to $\mathcal{Q}_{i-1}$, which is an *infeasible* package for $\mathcal{Q}_i$.

We measure the runtime *speedup* of preconditioning as the ratio of the running time of **NoStart** over **FeasibleStart** and **InfeasibleStart**, for feasible and infeasible starting packages, respectively. A speedup of 1 means that preconditioning has no effect on the running time. A speedup < 1 means that preconditioning led to worse performance and a speedup > 1 means that preconditioning improved the performance. Figure 5.5 shows the average speedup of **FeasibleStart** and **InfeasibleStart** across each query sequence, for the three types of generated sequences. Our results show that preconditioning does not consistently improve the performance of all queries. In fact, seeding the solver with an infeasible package can frequently lead to worse performance. On the other hand, **FeasibleStart** rarely hurts runtime performance, and can often help significantly—as much as 12x improvement in our experiment. This contrast between **FeasibleStart** and **InfeasibleStart** is intuitive: **Direct** needs to derive a solution that is (1) feasible and (2) has optimal objective value, so a seed that already satisfies the first condition is more likely to be useful.

Overall, the results of our empirical analysis indicate that preconditioning is a promising strategy for improving package query performance that merits further study.

Discussion

Our empirical study of preconditioning indicates that providing feasible packages as starting solutions can significantly speed up the computation of **Direct**. Future research could develop several ways in which a system could take advantage of this phenomenon. First, the system can maintain results of past queries in a solution pool that can be searched to identify good candidate starting packages for newly submitted queries. Second, it may be possible to construct simple feasible packages by executing a simplified package query, or even a set of traditional SQL queries. Furthermore, incremental evaluation can also directly benefit iterative query refinement (such as in data exploration), as results to previous queries are natural starting packages for subsequent ones.

5.4 Beyond packages: Data management for data-driven decision making

Package queries connect interesting business applications with the data used to make decisions, in a very natural way. While package queries capture many of the interesting decision-making problems businesses face today, there are more complex ones that fall outside of their scope. We believe that the lessons learned on package queries can fundamentally change the future of research for creating scalable systems for data-driven decisions making for larger and more diverse classes of problems. Decision making is a central component of nearly every aspect of our society. Modern applications require use of increasingly more data, rendering existing solutions inapplicable. As a result, approaches for decision making often simplify the problems so much that solutions are either infeasible

or highly inaccurate. One of the main goals of a database system is to allow classes of computational problems (i.e., "queries") to be easily expressed and efficiently executed regardless of the size of the input data and the availability of special hardware. With package queries, we allowed support for decision-making problems that can be expressed as integer linear programs. An important future direction is to enable other, more complex kinds of applications.

A Markov Decision Process (MDP) is a decision-making framework where a decision maker (agent) has to decide what actions to take at each time step (policy), given that actions lead to uncertain outcomes (rewards). The agent's objective is to maximize the expected future reward. Problems that can be modeled as MDPs include reinforcement learning, robot planning, and self-driving cars, just to name a few, and appear in a broad range of domains, including finance, investments, agriculture, robotics, etc. An important future goal is to create a new data-oriented system for efficiently and scalably solving MDPs. The main challenges include: simple and declarative languages, close to the data, for expressing large and complex state spaces for MDPs; efficient and scalable algorithms for solving large MDPs [109]; support for evolving MDPs, where the agent's environment change over time, and the underlining MDP needs to be updated.

Data management technology, and in particular the work presented in this thesis, can potentially lead to new solutions in some areas of robotics as well [109]. Robotics pose unique data management challenges. Consider a robot that needs to act autonomously in the world for long periods of time, without human intervention or without access to powerful computers. For example, a robot operating on Mars has to wait for several minutes before receiving a response back from Earth, due to the sheer distance between the two planets. Further, the sheer amount of sensor data of modern robotics applications are very demanding. For example, an autonomous vehicle, can produce more than 30 GB/hour

of data [106]. For a robot to be autonomous, it must be equipped with the ability to store a large amount of sensor data (from its walks, interactions, and findings), create complex knowledge representations about the world, and use it to solve large analytics, optimization and decision-making problems, in order to make autonomous plans for the future.

As machine-learned software becomes more ubiquitous and accessible to decision makers who can impact our society, and robots more and more part of our everyday life, there will be a growing need for systems that ensure *fair*, *responsible*, and *sustainable* solutions. There is growing interest to develop Socially Responsible Investments (SRI) [1, 2], in line with the 2015 UN Sustainable Development Goals [3]. For example, in SRI, investors are not only interested in reducing the risk of a loss, but also that their investments meet the ESG [157] (Environmental, Social and Governance) standards. Machine Learning models can exhibit undesirable behavior [154], from financial loss and unfair classification and predictions, to automated systems and robots that could potentially harm humans. Typically, a lot of training data is required to increase the confidence on the good behavior of the resulting model. However, training a model under complex constraints using very large datasets can be extremely inefficient or even prohibitively expensive. Thus, an important question is whether the techniques developed in this thesis, such as divide-and-conquer methodologies to break down large ILPs, reducing the data movement, and summarization techniques, can be adapted to efficiently train a Machine Learning model that requires complex constraints on large training sets.

CHAPTER 6

RELATED WORK

Being highly interdisciplinary, package queries connect several areas of Computer Science, such as Database Systems, Approximation Theory, and Query Languages, and areas of Operations Research, such as Integer and Stochastic Programming, Constrained Optimization, as well as areas at the intersection between the two, such as Business Analytics, Planning, Robotics, and Artificial Intelligence.

Package recommendations. Package or set-based recommendation systems [160, 161] are closely related to package queries. A package recommendation system presents users with interesting sets of items that satisfy some global conditions. These systems are usually driven by specific application scenarios. For instance, in the CourseRank [115] system, the items to be recommended are university *courses*, and the types of constraints are course-specific (e.g., prerequisites, incompatibilities, etc.). *Satellite packages* [13] are sets of items, such as smartphone accessories, that are compatible with a "central" item, such as a smartphone. Other related problems in the area of package recommendations are *team formation* [94, 14], and recommendation of *vacation* and *travel packages* [45]. Queries expressible in these frameworks are also expressible in PaQL, but the opposite does not hold. The complexity of set-based package recommendation problems is studied in [50], where the authors show that the data complexity of computing top-k packages [162] with a conjunctive query language is $\mathrm{FP}^{\mathrm{NP}}$-complete.

Semantic window queries. Packages are also related to *semantic windows* [83]. A semantic window defines a contiguous subset of a grid-partitioned space with certain global properties. For instance, astronomers can partition the night sky into a grid, and look for regions of the sky whose overal brightness is above a specific threshold. If the grid cells are precomputed and stored into an input relation, these queries can be expressed in PaQL by adding a global constraint (besides the brightness requirement) that ensures that all cells in a package must form a contiguous region in the grid space. Packages, however, are more general than semantic windows because they allow regions to be non-contiguous, or to contain gaps. Moreover, package queries also allow optimization criteria, which are not expressible in semantic window queries. A recent extension to methods for answering semantic window queries is Searchlight [84], which expresses these queries in the form of constraint programs. Searchlight uses in-memory synopses to quickly estimate aggregate values of contiguous regions. However, it does not support synopses for non-contiguous regions, and thus it cannot solve arbitrary package queries.

Iceberg queries. Iceberg queries are SQL group-by aggregation queries with a highly selective HAVING clause [57, 112, 93]. Package queries are much more powerful than iceberg queries, which cannot return packages of items, (they can only return group-by aggregates), and cannot express optimization objectives.

How-to queries. Package queries are related to how-to queries [104], as they both use an ILP formulation to translate the original queries. However, there are several major differences between package queries and how-to queries: package queries specify tuple collections, whereas how-to queries specify updates to underlying datasets; package queries allow a tuple to appear multiple times in a package result, while how-to queries do not model repetitions; PaQL is SQL-based whereas how-to queries use a variant of Datalog;

PaQL supports arbitrary Boolean formulas in the SUCH THAT clause, whereas how-to queries can only express conjunctive conditions.

Answer set programming. In answer set programming (ASP) [22, 63], logic programs follow a Datalog-like syntax with extended functionalities. ASP, extended with arithmetic, is able to express package queries, and packages can be seen as stable models of ASP programs. While ASP can express packages, SQL-based PaQL offers a more natural extension for most relational systems. More importantly, state-of-the-art ASP solvers, like Clingo [63] from the Potassco bundle, are not yet able to scale package computation to reasonable data sizes. We observed these shortcomings by running ASP problems for our Galaxy queries: the ASP solver did not scale to more than a few dozens of tuples, while ILP solvers scale up to millions of tuples.

Constraint query languages. The principal idea of constraint query languages (CQL) [87] is that a tuple can be generalized as a conjunction of constraints over variables. This principle is very general and creates connections between declarative database languages and constraint programming. However, prior work focused on expressing constraints over tuple values, rather than over sets of tuples. In this light, PaQL follows a similar approach to CQL by embedding in a declarative query language methods that handle higher-order constraints. However, our package query engine design allows for the direct use of ILP solvers as black-box components, automatically transforming problems and solutions from one domain to the other. In contrast, CQL needs to appropriately adapt the algorithms themselves between the two domains, and existing literature does not provide this adaptation for the constraint types in PaQL.

ILP approximations. There exists a large body of research in approximation algorithms for problems that can be modeled as integer linear programs. A typical approach is *linear programming relaxation* [159] in which the integrality constraints are dropped and variables are free to take on real values. These methods are usually coupled with *rounding* techniques that transform the real solutions to integer solutions with provable approximation bounds. None of these methods, however, can solve package queries on a large scale because they all assume that the LP solver is used on the entire problem. Another common approach to approximate a solution to an ILP problem is the *primal-dual method* [67]. All primal-dual algorithms, however, need to keep track of all primal and dual variables and the coefficient matrix, which means that none of these methods can be employed on large datasets. On the other hand, rounding techniques and primal-dual algorithms could potentially benefit from the **SketchRefine** algorithm to break down their complexity on very large datasets.

Approximations to subclasses of package queries. Like package queries, *optimization under parametric aggregation constraints* (OPAC) queries [69] can construct sets of tuples that collectively satisfy summation constraints. However, existing solutions to OPAC queries have several shortcomings: (1) they do not handle tuple repetitions; (2) they only address *multi-attribute knapsack queries*, a subclass of package queries where all global constraints are of the form $\mathsf{SUM}() \leq c$, with a $\mathsf{MAXIMIZE\ SUM}()$ objective criterion; (3) they may return infeasible packages; (4) they are conceptually different from **SketchRefine**, as they generate approximate solutions in a pre-processing step, and packages are simply retrieved at query time using a multi-dimensional index. In contrast, **SketchRefine** does not require pre-computation of packages. Package queries also encompass *submodular* optimization queries, whose recent approximate solutions use greedy distributed algorithms [105].

Probabilistic databases and package queries. *Probabilistic databases* [43, 149] have focused mainly on modeling discrete data uncertainty; the *Monte Carlo Database* (MCDB) [80] supports arbitrary uncertainty, via VG functions. Probabilistic databases support SQL queries, but lack support for optimization. *Package query engines* [23, 142] offer support only for deterministic optimization.

Stochastic optimization. *Stochastic optimization* [47] studies approximations for stochastic constraints and objectives. Probabilistic constraints are very hard to handle in general, because the feasible region of the inner constraint may be non-convex [47, 7, 31, 35, 111, 51, 100]. In this work, we study stochastic optimization problems with objective functions and constraints defined in terms of linear functions of the tuple attributes.

Our **Naïve** method is derived from the numerous "scenario approximations" from the stochastic programming literature [47, 85, 31, 35, 111, 99, 33, 110]. Choosing the number of scenarios (M) a priori is one of the most studied problems. Campi et al. [35] show that the optimal solution of a Monte Carlo formulation that satisfies exactly M i.i.d. scenarios is feasible with probability at least δ if $M \geq \frac{2}{1-p_j}\left(\ln\left(\frac{1}{1-\delta}\right)+N\right)$. A-priori bounds quickly become impractical in a database setting, where N is also the number of tuples, and thus typically large. For example, with a table of size $N = 50{,}000$, $p_j = 0.9$, $\delta = 0.95$, at least $M \geq 1{,}000{,}060$ scenarios must be generated and all satisfied.

Scenario removal studies techniques for removing scenarios after sampling [33, 55, 88, 99, 30]. Empirically, these methods generally provide a reduction factor of only 50% or less, which is insufficient for our setting. Our α-summary can be viewed as removing $100(1-\alpha)\%$ of the scenarios, where α is usually very small (below 0.01); not only do we remove scenarios, but we replace them with conservative summaries.

Similar to our setting, *distributionally robust optimization* (DRO) [71, 49, 92] attempts to mitigate the optimizer's curse when the uncertainty distribution is unknown but is assumed to lie in some set of candidate distributions; the original probability constraints are replaced with worst-case probability constraints based on this set. In contrast, **SummarySearch** uses deterministic worst-case constraints, which are simpler and avoid assumptions on the uncertainty distribution. DRO methods also show limited scalability in the number of variables N, e.g., N is at most 20 in the experiments in [92].

The goal of *wait-and-judge optimization* [34, 36] is to perform a-posteriori feasibility analysis. Existing approaches help provide bounds on the quality of a solution, but do not provide algorithms that dynamically adapt in response to poor solutions. **SummarySearch**, instead, adjusts the conservativeness of the summaries to obtain feasible solutions with minimum computational cost. **SummarySearch** can potentially use wait-and-judge during out-of-sample validation to decide when to stop increasing the number of scenarios.

Document summarization. In SuDocu (Section 5.2.4), we focus on producing a personalized *extractive* summary of each document within a collection of documents. Such a summary directly selects sentences from the document to form the summary. (In contrast, *abstractive* summarization, which synthesizes new sentences that embody a holistic understanding of the document, is a much harder task; even state-of-the-art deep learning methods struggle to produce human-readable summaries [141].) The key issues are: how to (1) express the user's intent, and (2) select the set of sentences that, collectively, best satisfy the user's intent.

In *query-based* summarization, users specify their intent in the form of an unstructured query—typically, a natural language question. For example, the question "What are some interesting places?" is very subjective, as different people consider different places as

interesting. For a nature enthusiast, parks, lakes, oceans, and mountains are interesting; for an art enthusiast, museums, concerts, and plays are interesting. Some approaches use hints that represent user interest. Such hints take different forms, such as user-provided annotations [108], vision-based eye-tracking [164], user history and collaborative social influences [129], and so on. SuDocu allows the user to provide precise and concrete examples of the type of summaries they want, and does not require large training data. A possible way to adapt query-based summarization for example-driven summarization is to infer the underlying natural-language query from the example summaries, and then use an existing tool. However, computers understand structured queries with clear semantics, which can easily be constructed from examples, much better than natural language queries, so an example-based approach is both simpler and more accurate.

Early approaches to sentence selection would score each sentence based on some criteria and return the top-k sentences as a summary. This would often lead to the inclusion of redundant sentences. To tackle the issue of redundancy, later work [70] followed an ad hoc iterative greedy approach, leading to suboptimal summaries. Alternative approaches based on topic modeling identify a set of topics that the user cares about (perhaps extracted from examples) and then pick the best sentence per topic to construct the summary. However, such a summary can also be suboptimal, as sentences often cover multiple topics; a sentence that is not top-scoring in any single topic, but covers multiple topics well, might be excluded from the summary. While some approaches try to iteratively refine the summary quality [8], they are mostly based on heuristic approaches, e.g., A* search, that still do not guarantee optimality.

A shortcoming of the foregoing sentence-selection approaches is that they consider candidate sentences in isolation, rather than trying to select a set of sentences that collectively form a good summary. The problem of selecting the best *set* of sentences can

be formulated as an integer program. Lin and Bilmes [98] provide an integer programming formulation with constraints and objectives involving general sentence score, diversity, and summary length, but with no connection to the user-provided examples. In contrast, our formulation can capture the summarization intent from the example summaries using constraints on how much each topic should be "covered" by the summary; roughly speaking, the coverage should resemble that of the user-provided examples. Also, because of the combinatorially large number of possible summaries, the formulation in [98] cannot generally scale to large dataset sizes. We use our previously-developed **SketchRefine** algorithm [25] to scale the resulting integer linear program to very large datasets.

Querying by example. Query by Output (QBO) [155] and Synthesizing View Definitions (SVD) [44] focus on the problem of learning a query Q from a given input database D, and an output view V. These works differ from our work SuDocu (Section 5.2.4) and QSynth (Section 5.3): both QBO and SVD learn simpler database queries that return individual tuples and not sets of tuples. There has been no research on learning packages from examples. A survey of active learning can be found in [138]. Other research works in query synthesis include [40, 114, 39].

Clustering and information retrieval. Clustering is used in information retrieval under the "cluster hypothesis": Documents in the same clusters tend to behave similarly with regards to the same queries [103]. Clustering is often used after retrieving an initial set of relevant documents to improve the results. A set of documents from the cluster of some of the retrieved documents is added to the ranking even if they have low scores (according to the ranking metric). In this work, we utilize clustering to group together tuples and form answer packages.

Planning. A Markov Decision Process (MDP) is a decision-making framework where a decision maker (agent) has to decide what actions to take at each time step (policy), given that actions lead to uncertain outcomes (rewards) [109]. The agent's objective is to maximize the expected future reward. Problems that can be modeled as MDPs include reinforcement learning, robot planning, and self-driving cars, just to name a few, and appear in a broad range of domains, including finance, investments, agriculture, robotics, etc.

The desire to solve large MDPs is not new, and techniques for doing so generally adopt one of three approaches. First, there are *approximate solvers* that use dynamic programming, such as value or policy iteration [16, 125], and linear programming [68, 117, 124, 102]. Second, some methods compute *partial policies* on a subset of the ground states and re-plan if the agent encounters a state for which the partial policy is undefined [146, 118]. Third, optimal policies are computed on *abstractions* of the original problem, where there is a surjective mapping from the original ground states to the abstract states [97]. Using abstractions to reduce the size of a problem is a natural and popular approach to solving large MDPs. The quality of these policies depends heavily on the abstraction scheme, and many abstraction methods have been proposed. Some strict definitions include bisimulation [65], statistical bisimulation [60], and bounded MDPs [48]. Abstractions based on homomorphisms [128, 20] and generic change of basis have also been proposed [167]. Abstractions also support hierarchical systems, such as MAXQ [52] and task-relevant partitions [131]. Abstractions for continuous variables [96] and across time [73] have been examined as well. Some work even uses temporal abstractions derived from analytically computed landmarks to summarize policies for SSPs [148]. Another form of abstraction is determinization, or its more general form, reduced models, which form abstractions over action outcomes [119, 134]. Computing partial policies is an approach with a history

of success. FF-Replan [166], a remarkably simple yet effective algorithm for planning in MDPs, works by determinizing an MDP, constructing a plan to the determinization, and re-planning if the agent reaches an unexpected state. Recently, Soft-FLARES [120] achieved impressive results on large stochastic shortest path (SSP) problems, a subclass of MDPs, by probabilistically labeling ϵ-consistent state values within a horizon. There has also been work on designing meta-level controllers to reason about when to expand additional states within an MDP while solving for a partial policy [10]. Partial policies over *actions* have even been explored in an effort to bias Monte Carlo tree search over policies online [122].

CHAPTER 7

CONCLUSION

This thesis presented the first complete and efficient data management system to support *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets, allowing the declarative specification and efficient evaluation of a significant class of constrained optimization problems–integer programs–within a relational database. Package queries are a unified solution to enable declarative and scalable prescriptive analytics close to the data. This dissertation presented solutions for expressing package queries, and for evaluating them in an efficient and effective way, when the data is deterministic or uncertain, and in the presence of large amounts of data. Further, it also showcased important applications of package queries in a variety of domains, from healthcare, to finance and science, supported through the development of sophisticated user interfaces and demonstration scenarios.

Package queries connect interesting business applications with the data used to make decisions, in a very natural way. While package queries capture many of the interesting decision-making problems businesses face today, there are more complex ones that fall outside of their scope. We believe that the lessons learned on package queries can fundamentally change the future of research for creating scalable systems for data-driven decisions making for larger and more diverse classes of problems. Decision making is a central component of nearly every aspect of our society. Modern applications require use of increasingly more data, rendering existing solutions inapplicable. As a result, approaches

for decision making often simplify the problems so much that solutions are either infeasible or highly inaccurate. One of the main goals of a database system is to allow classes of computational problems (i.e., "queries") to be easily expressed and efficiently executed regardless of the size of the input data and the availability of special hardware. With package queries, we allowed support for decision-making problems that can be expressed as integer linear programs. But data-driven decision making has a plethora of different applications, each using with different data types, and with different types of assumptions, ideal usage patterns, constraints and objectives. A unique system that can address *all* kinds of decision-making applications is perhaps utopical. But we believe it is possible to identify classes of applications sharing similar settings and build systems to support each individual class. Package queries are an example of this, and this thesis demonstrates that it is possible to address this class of queries successfully. In our future work, we plan to identify new classes of data-driven decision-making applications and build new systems to support them.

BIBLIOGRAPHY

[1] ESG investing: Where your money can reflect what matters to you. `https://investor.vanguard.com/investing/esg/`.

[2] Principles of Responsible Investment. `https://www.unpri.org/`.

[3] The SDG Investment Case. `https://www.unpri.org/sdgs/the-sdg-investment-case/303.article`.

[4] Sudocu: `http://sudocu.cs.umass.edu/demo/`, 2020.

[5] Abouzied, Azza, Angluin, Dana, Papadimitriou, Christos H., Hellerstein, Joseph M., and Silberschatz, Avi. Learning and verifying quantified boolean queries by example. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013* (2013), pp. 49–60.

[6] Abouzied, Azza, Hellerstein, Joseph, and Silberschatz, Avi. Dataplay: Interactive tweaking and example-driven correction of graphical database queries. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (2012), ACM, pp. 207–218.

[7] Ahmed, Shabbir, and Shapiro, Alexander. Solving chance-constrained stochastic programs via sampling and integer programming. In *State-of-the-Art Decision-Making Tools in the Information-Intensive Age*. Informs, 2008, pp. 261–269.

[8] Aker, Ahmet, Cohn, Trevor, and Gaizauskas, Robert. Multi-document summarization using A* search and discriminative training. In *EMNLP* (2010), pp. 482–491.

[9] Alagoz, Oguzhan, Schaefer, Andrew J., and Roberts, Mark S. *Optimizing Organ Allocation and Acceptance*. Springer, Boston, MA, 2009, pp. 1–24.

[10] Alexander, George, Raja, Anita, and Musliner, David J. Controlling deliberation in a Markov decision process-based agent. In *International Joint Conference on Autonomous Agents and Multiagent Systems* (2008), Citeseer, pp. 461–468.

[11] Baeza-Yates, Ricardo A., and Ribeiro-Neto, Berthier A. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.

[12] Banerjee, Arindam, Bandyopadhyay, Tathagata, and Acharya, Prachi. Data analytics: Hyped up aspirations or true potential? *Vikalpa 38*, 4 (2013), 1–12.

[13] Basu Roy, Senjuti, Amer-Yahia, Sihem, Chawla, Ashish, Das, Gautam, and Yu, Cong. Constructing and exploring composite items. In *SIGMOD* (2010), pp. 843–854.

[14] Baykasoglu, Adil, Dereli, Turkay, and Das, Sena. Project team selection using fuzzy optimization approach. *Cybernetic Systems 38*, 2 (2007), 155–185.

[15] Bentley, Jon Louis. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[16] Bertsekas, Dimitri P. Approximate policy iteration: A survey and some new methods. *Journal of Control Theory and Applications 9*, 3 (2011), 310–335.

[17] Bertsimas, Dimitris, and Kallus, Nathan. From predictive to prescriptive analytics. *arXiv preprint arXiv:1402.5481* (2014).

[18] Bienstock, Daniel, Chertkov, Michael, and Harnett, Sean. Chance-constrained optimal power flow: Risk-aware network control under uncertainty. *SIAM Review 56*, 3 (2014), 461–495.

[19] Bisschop, Johannes. *AIMMS Optimization Modeling*. Paragon Decision Technology, 2006.

[20] Biza, Ondrej, and Platt, Robert. Online abstraction with MDP homomorphisms for deep learning. *arXiv preprint arXiv:1811.12929* (2018).

[21] Blei, David M., Ng, Andrew Y., and Jordan, Michael I. Latent dirichlet allocation. *J. Mach. Learn. Res. 3* (2003), 993–1022.

[22] Bonatti, Piero, Calimeri, Francesco, Leone, Nicola, and Ricca, Francesco. A 25-year perspective on logic programming. Springer-Verlag, Berlin, Heidelberg, 2010, ch. Answer Set Programming, pp. 159–182.

[23] Brucato, Matteo, Abouzied, Azza, and Meliou, Alexandra. Package queries: efficient and scalable computation of high-order constraints. *The VLDB Journal* (Oct 2018).

[24] Brucato, Matteo, Abouzied, Azza, and Meliou, Alexandra. Scalable computation of high-order optimization queries. *Commun. ACM 62*, 2 (Jan. 2019), 108–116.

[25] Brucato, Matteo, Beltran, Juan Felipe, Abouzied, Azza, and Meliou, Alexandra. Scalable package queries in relational database systems. *PVLDB 9*, 7 (2016), 576–587.

[26] Brucato, Matteo, Mannino, Miro, Abouzied, Azza, Haas, Peter J, and Meliou, Alexandra. sPaQLTooLs: a stochastic package query interface for scalable constrained optimization. *Proceedings of the VLDB Endowment 13*, 12 (2020), 2881–2884.

[27] Brucato, Matteo, Ramakrishna, Rahul, Abouzied, Azza, and Meliou, Alexandra. PackageBuilder: From tuples to packages. *PVLDB 7*, 13 (2014), 1593–1596.

[28] Brucato, Matteo, Yadav, Nishad, Abouzied, Azza, Haas, Peter J., and Meliou, Alexandra. Stochastic package queries in probabilistic databases. In *SIGMOD* (2020).

[29] Cai, Zhuhua, Vagena, Zografoula, Perez, Luis, Arumugam, Subramanian, Haas, Peter J, and Jermaine, Christopher. Simulation of database-valued Markov chains using SimSQL. In *ACM SIGMOD* (2013), pp. 637–648.

[30] Calafiore, Giuseppe, and Campi, M.C. Uncertain convex programs: randomized solutions and confidence levels. *Mathematical Programming 102*, 1 (Jan 2005), 25–46.

[31] Calafiore, Giuseppe, and Dabbene, Fabrizio. *Probabilistic and randomized methods for design under uncertainty*. Springer, 2006.

[32] Calafiore, Giuseppe C, and Campi, Marco C. The scenario approach to robust control design. *IEEE Transactions on Automatic Control 51*, 5 (2006), 742–753.

[33] Campi, Marco C, and Garatti, Simone. A sampling-and-discarding approach to chance-constrained optimization: feasibility and optimality. *Journal of Optimization Theory and Applications 148*, 2 (2011), 257–280.

[34] Campi, Marco C, and Garatti, Simone. Wait-and-judge scenario optimization. *Mathematical Programming 167*, 1 (2018), 155–189.

[35] Campi, Marco C., Garatti, Simone, and Prandini, Maria. The scenario approach for systems and control design. *Annual Reviews in Control 33*, 2 (2009), 149 – 157.

[36] Campi, Marco Claudio, Garatti, Simone, and Ramponi, Federico Alessandro. A general scenario theory for nonconvex optimization and decision making. *IEEE Transactions on Automatic Control 63*, 12 (2018), 4067–4078.

[37] Campi, MC, and Calafiore, G. Decision making in an uncertain environment: the scenario-based optimization approach. *Multiple Participant Decision Making* (2004), 99–111.

[38] Chen, Der-San, Batson, Robert G, and Dang, Yu. *Applied integer programming: modeling and solution*. John Wiley & Sons, 2011.

[39] Cheung, Alvin, Solar-Lezama, Armando, and Madden, Samuel. Inferring sql queries using program synthesis. *CoRR abs/1208.2013* (2012).

[40] Cheung, Alvin, Solar-Lezama, Armando, and Madden, Samuel. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2013), pp. 3–14.

[41] Clare, Gillian, and Richards, Arthur. Air traffic flow management under uncertainty: application of chance constraints. In *Proc. 2nd Intl. Conf. Application and Theory of Automation in Command and Control Systems* (2012), IRIT Press, pp. 20–26.

[42] Cook, William, and Hartmann, M. On the complexity of branch and cut methods for the traveling salesman problem. *Polyhedral Combinatorics 1* (1990), 75–82.

[43] Dalvi, Nilesh, and Suciu, Dan. Efficient query evaluation on probabilistic databases. *The VLDB Journal–The International Journal on Very Large Data Bases 16*, 4 (2007), 523–544.

[44] Das Sarma, Anish, Parameswaran, Aditya, Garcia-Molina, Hector, and Widom, Jennifer. Synthesizing view definitions from data. In *Proceedings of the 13th International Conference on Database Theory (ICDT)* (2010), pp. 89–103.

[45] De Choudhury, Munmun, Feldman, Moran, Amer-Yahia, Sihem, Golbandi, Nadav, Lempel, Ronny, and Yu, Cong. Automatic construction of travel itineraries using social breadcrumbs. In *HyperText* (2010), pp. 35–44.

[46] De Choudhury, Munmun, Feldman, Moran, Amer-Yahia, Sihem, Golbandi, Nadav, Lempel, Ronny, and Yu, Cong. Automatic construction of travel itineraries using social breadcrumbs. In *HyperText* (2010), pp. 35–44.

[47] de Mello, Tito Homem, and Bayraksan, Güzin. Monte carlo sampling-based methods for stochastic optimization. *Surveys in Operations Research and Management Science 19*, 1 (2014), 56 – 85.

[48] Dean, Thomas L, Givan, Robert, and Leach, Sonia. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. *arXiv preprint arXiv:1302.1533* (1997).

[49] Delage, Erick, and Ye, Yinyu. Distributionally robust optimization under moment uncertainty with application to data-driven problems. *Operations research 58*, 3 (2010), 595–612.

[50] Deng, Ting, Fan, Wenfei, and Geerts, Floris. On the complexity of package recommendation problems. In *PODS* (2012), pp. 261–272.

[51] Dentcheva, Darinka. Optimization models with probabilistic constraints. In *Probabilistic and randomized methods for design under uncertainty*. Springer, 2006, pp. 49–97.

[52] Dietterich, Thomas G. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research 13* (2000), 227–303.

[53] Dong, Xin Luna, Halevy, Alon, and Yu, Cong. Data integration with uncertainty. *The VLDB Journal 18*, 2 (2009), 469–500.

[54] Du Toit, Noel E, and Burdick, Joel W. Probabilistic collision checking with chance constraints. *IEEE Transactions on Robotics 27*, 4 (2011), 809–815.

[55] Dupačová, Jitka, Gröwe-Kuska, Nicole, and Römisch, Werner. Scenario reduction in stochastic programming. *Mathematical programming 95*, 3 (2003), 493–511.

[56] Ester, Martin, Kriegel, Hans-Peter, Sander, Jörg, and Xu, Xiaowei. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD* (1996), pp. 226–231.

[57] Fang, Min, Shivakumar, Narayanan, Garcia-Molina, Hector, Motwani, Rajeev, and Ullman, Jeffrey D. Computing iceberg queries efficiently. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA* (1998), pp. 299–310.

[58] Fariha, Anna, and Meliou, Alexandra. Example-driven query intent discovery: Abductive reasoning using semantic similarity. *PVLDB 12*, 11 (2019), 1262–1275.

[59] Fernandes, Kevin, Brucato, Matteo, Ramakrishna, Rahul, Abouzied, Azza, and Meliou, Alexandra. Packagebuilder: Querying for packages of tuples. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 1613–1614.

[60] Ferns, Norm, Panangaden, Prakash, and Precup, Doina. Metrics for finite Markov decision processes. In *Conference on Uncertainty in Artificial Intelligence* (2004), vol. 4, pp. 162–169.

[61] Finkel, Raphael A., and Bentley, Jon Louis. Quad trees a data structure for retrieval on composite keys. *Acta informatica 4*, 1 (1974), 1–9.

[62] GDAL - Geospatial Data Abstraction Library. `http://www.gdal.org/`.

[63] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)* (2014), M. Leuschel and T. Schrijvers, Eds., vol. arXiv:1405.3694v1. Theory and Practice of Logic Programming, Online Supplement.

[64] Geng, Na, Xie, Xiaolan, and Zhang, Zheng. Addressing healthcare operational deficiencies using stochastic and dynamic programming. *International Journal of Production Research 57*, 14 (2019), 4371–4390.

[65] Givan, Robert, Dean, Thomas, and Greig, Matthew. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence 147*, 1-2 (2003), 163–223.

[66] GNU Bison. `https://www.gnu.org/software/bison/`.

[67] Goemans, Michel X, and Williamson, David P. The primal-dual method for approximation algorithms and its application to network design problems. *Approximation algorithms for NP-hard problems* (1997), 144–191.

[68] Guestrin, Carlos, Koller, Daphne, Parr, Ronald, and Venkataraman, Shobha. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research 19* (2003), 399–468.

[69] Guha, Sudipto, Gunopulos, Dimitrios, Koudas, Nick, Srivastava, Divesh, and Vlachos, Michail. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB* (2003), pp. 778–789.

[70] Haghighi, Aria, and Vanderwende, Lucy. Exploring content models for multi-document summarization. In *NAACL* (2009), pp. 362–370.

[71] Hanasusanto, Grani A, Roitch, Vladimir, Kuhn, Daniel, and Wiesemann, Wolfram. A distributionally robust perspective on uncertainty quantification and chance constrained programming. *Mathematical Programming 151*, 1 (2015), 35–62.

[72] Hartigan, John A, and Wong, Manchek A. Algorithm as 136: A k-means clustering algorithm. *Applied statistics* (1979), 100–108.

[73] Hauskrecht, Milos, Meuleau, Nicolas, Kaelbling, Leslie Pack, Dean, Thomas L, and Boutilier, Craig. Hierarchical solution of Markov decision processes using macro-actions. *arXiv preprint arXiv:1301.7381* (2013).

[74] Hoeffding, Wassily. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association 58*, 301 (1963), 13–30.

[75] Hong, L Jeff, Hu, Zhaolin, and Liu, Guangwu. Monte Carlo methods for value-at-risk and conditional value-at-risk: a review. *ACM Trans. Modeling and Computer Simulation 24*, 4 (2014), 22.

[76] IBM CPLEX Optimization Studio. `http://www.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[77] Jagadish, H. V., Chapman, Adriane, Elkiss, Aaron, Jayapandian, Magesh, Li, Yunyao, Nandi, Arnab, and Yu, Cong. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2007), pp. 13–24.

[78] Jagadish, H. V., Nandi, Arnab, and Qian, Li. Organic databases. In *Proceedings of the 7th International Conference on Databases in Networked Information Systems (DNIS)* (2011), Springer-Verlag, pp. 49–63.

[79] Jampani, Ravi, Xu, Fei, Wu, Mingxi, Perez, Luis Leopoldo, Jermaine, Chris, and Haas, Peter J. The Monte Carlo database system: Stochastic analysis close to the data. *ACM Trans. Database Syst. 36*, 3 (2011), 18:1–18:41.

[80] Jampani, Ravi, Xu, Fei, Wu, Mingxi, Perez, Luis Leopoldo, Jermaine, Christopher, and Haas, Peter J. MCDB: A Monte Carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 687–700.

[81] Johnson, Noah, Near, Joseph P, and Song, Dawn. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment 11*, 5 (2018), 526–539.

[82] Jorion, Philippe, et al. *Financial Risk Manager Handbook*, vol. 406. John Wiley & Sons, 2007.

[83] Kalinin, Alexander, Cetintemel, Ugur, and Zdonik, Stan. Interactive data exploration using semantic windows. In *SIGMOD* (2014), pp. 505–516.

[84] Kalinin, Alexander, Çetintemel, Ugur, and Zdonik, Stanley B. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB 8*, 10 (2015), 1094–1105.

[85] Kall, Peter, Wallace, Stein W, and Kall, Peter. *Stochastic Programming*. Springer, 1994.

[86] Kandel, Sean, Paepcke, Andreas, Hellerstein, Joseph, and Heer, Jeffrey. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), ACM, pp. 3363–3372.

[87] Kanellakis, PC, Kuper, GM, and Revesz, PZ. Constraint query languages. *Journal of Computer and System Sciences 1*, 51 (1995), 26–52.

[88] Karuppiah, Ramkumar, Martin, Mariano, and Grossmann, Ignacio E. A simple heuristic for reducing the number of scenarios in two-stage stochastic programming. *Computers & Chemical Engineering 34*, 8 (2010), 1246–1255.

[89] Kaufman, Leonard, and Rousseeuw, Peter J. *Finding groups in data: an introduction to cluster analysis*, vol. 344. John Wiley & Sons, 2009.

[90] Kennedy, Oliver, and Koch, Christoph. Pip: A database system for great and small expectations. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (2010), IEEE, pp. 157–168.

[91] Kibriya, Ashraf M., and Frank, Eibe. An empirical comparison of exact nearest neighbour algorithms. In *Knowledge Discovery in Databases: PKDD 2007, 11th European Conference on Principles and Practice of Knowledge Discovery in Databases, Warsaw, Poland, September 17-21, 2007, Proceedings* (2007), pp. 140–151.

[92] Lam, Henry, and Li, Fengpei. Sampling uncertain constraints under parametric distributions. In *2018 Winter Simulation Conference (WSC)* (2018), IEEE, pp. 2072–2083.

[93] Laporte, Marc, Novelli, Noel, Cicchetti, Rosine, and Lakhal, Lotfi. Computing full and iceberg datacubes using partitions. In *Foundations of Intelligent Systems, 13th International Symposium, ISMIS 2002, Lyon, France, June 27-29, 2002, Proceedings* (2002), pp. 244–254.

[94] Lappas, Theodoros, Liu, Kun, and Terzi, Evimaria. Finding a team of experts in social networks. In *SIGKDD* (2009), pp. 467–476.

[95] Letsche, Todd A., and Berry, Michael W. Large-scale information retrieval with latent semantic indexing. *Inf. Sci. 100*, 1-4 (1997), 105–137.

[96] Li, Lihong, and Littman, Michael L. Lazy approximation for solving continuous finite-horizon MDPs. In *AAAI Conference on Artificial Intelligence* (2005), vol. 5, pp. 1175–1180.

[97] Li, Lihong, Walsh, Thomas J, and Littman, Michael L. Towards a unified theory of state abstraction for MDPs. In *International Symposium on Artificial Intelligence and Mathematics* (2006).

[98] Lin, Hui, and Bilmes, Jeff. Multi-document summarization via budgeted maximization of submodular functions. In *NAACL* (2010), pp. 912–920.

[99] Luedtke, James, and Ahmed, Shabbir. A sample approximation approach for optimization with probabilistic constraints. *SIAM Journal on Optimization 19*, 2 (2008), 674–699.

[100] Luedtke, James, Ahmed, Shabbir, and Nemhauser, George L. An integer programming approach for linear programs with probabilistic constraints. *Mathematical Programming 122*, 2 (Apr 2010), 247–272.

[101] Makuch, William M., Dodge, Jeffrey L., Ecker, Joseph G., Granfors, Donna C., and Hahn, Gerald J. Managing consumer credit delinquency in the us economy: A multi-billion dollar management science application. *Interfaces 22*, 1 (1992), 90–109.

[102] Malek, Alan, Abbasi-Yadkori, Yasin, and Bartlett, Peter. Linear programming for large-scale Markov decision problems. In *International Conference on Machine Learning* (2014), pp. 496–504.

[103] Manning, Christopher D., Raghavan, Prabhakar, and Schütze, Hinrich. *Introduction to information retrieval*. Cambridge University Press, 2008.

[104] Meliou, Alexandra, and Suciu, Dan. Tiresias: The database oracle for how-to queries. In *SIGMOD* (2012), pp. 337–348.

[105] Mirzasoleiman, Baharan, Karbasi, Amin, Sarkar, Rik, and Krause, Andreas. Distributed submodular maximization: Identifying representative elements in massive data. In *NIPS* (2013).

[106] Moll, Oscar, Zalewski, Aaron, Pillai, Sudeep, Madden, Sam, Stonebraker, Michael, and Gadepally, Vijay. Exploring big volume sensor data with vroom. *Proceedings of the VLDB Endowment 10*, 12 (2017), 1973–1976.

[107] Montague, Mark H. Metasearch: Data Fusion for Document Retrieval. Tech. Rep. TR2002-424, Dartmouth College, Computer Science, Hanover, NH, May 2002.

[108] Móro, Róbert, et al. Personalized text summarization based on important terms identification. In *DEXA* (2012), pp. 131–135.

[109] Nashed, Samer B, Svegliato, Justin, Brucato, Matteo, Basich, Connor, Grupen, Rod, and Zilberstein, Shlomo. Solving Markov decision processes with partial state abstractions. In *Proceedings of the IEEE International Conference on Robotics and Automation* (2021).

[110] Nemirovski, Arkadi, and Shapiro, Alexander. Convex approximations of chance constrained programs. *SIAM Journal on Optimization 17*, 4 (2006), 969–996.

[111] Nemirovski, Arkadi, and Shapiro, Alexander. Scenario approximations of chance constraints. In *Probabilistic and randomized methods for design under uncertainty*. Springer, 2006, pp. 3–47.

[112] Ng, Raymond T., Wagner, Alan S., and Yin, Yu. Iceberg-cube computation with PC clusters. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001* (2001), pp. 25–36.

[113] Padberg, Manfred, and Rinaldi, Giovanni. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review 33*, 1 (1991), 60–100.

[114] Parameswaran, Aditya, Sarma, Anish Das, Garcia-Molina, Hector, Polyzotis, Neoklis, and Widom, Jennifer. Human-assisted graph search: It's okay to ask questions. *Proceedings of the VLDB Endowment (PVLDB) 4*, 5 (Feb. 2011), 267–278.

[115] Parameswaran, Aditya G., Venetis, Petros, and Garcia-Molina, Hector. Recommendation systems with complex constraints: A course recommendation perspective. *ACM TOIS 29*, 4 (2011), 1–33.

[116] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[117] Petrik, Marek, and Zilberstein, Shlomo. Constraint relaxation in approximate linear programs. In *Annual International Conference on Machine Learning* (2009), pp. 809–816.

[118] Pineda, Luis Enrique, Wray, Kyle Hollins, and Zilberstein, Shlomo. Fast SSP solvers using short-sighted labeling. In *AAAI Conference on Artificial Intelligence* (2017).

[119] Pineda, Luis Enrique, and Zilberstein, Shlomo. Planning under uncertainty using reduced models: Revisiting determinization. In *24th International Conference on Automated Planning and Scheduling* (2014), Citeseer.

[120] Pineda, Luis Enrique, and Zilberstein, Shlomo. Soft labeling in stochastic shortest path problems. In *International Conference on Autonomous Agents and Multiagent Systems* (2019), pp. 467–475.

[121] Pinel, Florian, and Varshney, Lav R. Computational creativity for culinary recipes. In *CHI* (2014), pp. 439–442.

[122] Pinto, Jervis, and Fern, Alan. Learning partial policies to speedup MDP tree search. In *Conference on Uncertainty in Artificial Intelligence* (2014), Citeseer, pp. 672–681.

[123] PostGIS. `http://postgis.net/`.

[124] Poupart, Pascal, Malhotra, Aarti, Pei, Pei, Kim, Kee-Eung, Goh, Bongseok, and Bowling, Michael. Approximate linear programming for constrained partially observable markov decision processes. In *AAAI Conference on Artificial Intelligence* (2015), vol. 1, pp. 3342–3348.

[125] Powell, Warren B. Perspectives of approximate dynamic programming. *Annals of Operations Research 241*, 1-2 (2016), 319–356.

[126] QGIS. `http://www.qgis.org/it/site/`.

[127] Railroad Diagram Generator. `https://www.bottlecaps.de/rr/ui`.

[128] Ravindran, Balaraman, and Barto, Andrew G. Model minimization in hierarchical reinforcement learning. In *International Symposium on Abstraction, Reformulation, and Approximation* (2002), Springer, pp. 196–211.

[129] Ren, Zhaochun, Liang, Shangsong, Meij, Edgar, and de Rijke, Maarten. Personalized time-aware tweets summarization. In *SIGIR* (2013), pp. 513–522.

[130] Ross, Sheldon M. *Introduction to probability models*. Academic press, 2014.

[131] Ruiken, Dirk, Liu, Tiffany Q, Takahashi, Takeshi, and Grupen, Roderic A. Reconfigurable tasks in belief-space planning. In *IEEE-RAS International Conference on Humanoid Robots* (2016), IEEE, pp. 1257–1263.

[132] Rushmeier, Russell A., and Kontogiorgis, Spyridon A. Advances in the optimization of airline fleet assignment. *Transportation Science 31*, 2 (1997), 159–169.

[133] Ruthven, Ian, and Lalmas, Mounia. A survey on the use of relevance feedback for information access systems. *Knowledge Eng. Review 18*, 2 (2003), 95–145.

[134] Saisubramanian, Sandhya, and Zilbertsein, Shlomo. Adaptive outcome selection for planning with reduced models. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (2019), IEEE, pp. 1655–1660.

[135] Salton, Gerard, Wong, A., and Yang, C. S. A vector space model for automatic indexing. *Commun. ACM 18*, 11 (1975), 613–620.

[136] Sauer, Otto A., Shepard, David M., and Mackie, T. Rock. Application of constrained optimization to radiotherapy planning. *Medical Physics 26*, 11 (1999), 2359–2366.

[137] Schrijver, Alexander. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[138] Settles, Burr. *Active Learning*, vol. 6. Morgan & Claypool Publishers, 2012.

[139] Shapiro, Alexander, Dentcheva, Darinka, and Ruszczyński, Andrzej. *Lectures on stochastic programming: modeling and theory*. SIAM, 2009.

[140] Shaw, Joseph A., and Fox, Edward A. Combination of multiple searches. In *Proceedings of The Third Text REtrieval Conference, TREC 1994, Gaithersburg, Maryland, USA, November 2-4, 1994* (1994), pp. 105–108.

[141] Shi, Tian, Keneshloo, Yaser, Ramakrishnan, Naren, and Reddy, Chandan K. Neural abstractive text summarization with sequence-to-sequence models. *arXiv preprint arXiv:1812.02303* (2018).

[142] Šikšnys, Laurynas, and Pedersen, Torben Bach. SolveDB: Integrating optimization problem solvers into SQL databases. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management* (2016), ACM, p. 14.

[143] Singhal, Amit. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull. 24*, 4 (2001), 35–43.

[144] Sivarajah, Uthayasankar, Kamal, Muhammad Mustafa, Irani, Zahir, and Weerakkody, Vishanth. Critical analysis of big data challenges and analytical methods. *Journal of Business Research 70* (2017), 263 – 286.

[145] Smith, James E, and Winkler, Robert L. The optimizer's curse: Skepticism and postdecision surprise in decision analysis. *Management Science 52*, 3 (2006), 311–322.

[146] Smith, Trey, and Simmons, Reid. Focused real-time dynamic programming for MDPs: Squeezing more out of a heuristic. In *AAAI Conference on Artificial Intelligence* (2006), pp. 1227–1232.

[147] sPaQLTooLs. `http://packagebuilder.cs.umass.edu/spaqltools`.

[148] Sreedharan, Sarath, Srivastava, Siddharth, and Kambhampati, Subbarao. TLdR: Policy summarization for factored SSP problems using temporal abstractions. In *International Conference on Automated Planning and Scheduling* (2020), vol. 30, pp. 272–280.

[149] Suciu, Dan, Olteanu, Dan, Ré, Christopher, and Koch, Christoph. Probabilistic databases, synthesis lectures on data management. *Morgan & Claypool* (2011).

[150] Terrer, J. M. Artacho, Benede, M. A. Nasarre, del Rio, E. Bernues, and Llanas, S. Cruz. A feasible application of constrained optimization in the IMRT system. *IEEE Transactions on Biomedical Engineering 54*, 3 (2007), 370–379.

[151] The Sloan Digital Sky Survey. `http://www.sdss.org/`.

[152] The Sloan Digital Sky Survey, Data Release 12. `http://cas.sdss.org/dr12/`.

[153] The TPC-H Benchmark. `http://www.tpc.org/tpch/`.

[154] Thomas, Philip S, da Silva, Bruno Castro, Barto, Andrew G, Giguere, Stephen, Brun, Yuriy, and Brunskill, Emma. Preventing undesirable behavior of intelligent machines. *Science 366*, 6468 (2019), 999–1004.

[155] Tran, Quoc Trung, Chan, Chee-Yong, and Parthasarathy, Srinivasan. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2009), pp. 535–548.

[156] US Geological Survey. `http://www.usgs.gov/`.

[157] van Duuren, Emiel, Plantinga, Auke, and Scholtens, Bert. Esg integration and the investment management process: Fundamental investing reinvented. *Journal of Business Ethics 138*, 3 (2016), 525–533.

[158] Vardi, Moshe Y. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing* (1982), ACM, pp. 137–146.

[159] Williamson, David P, and Shmoys, David B. *The design of approximation algorithms*. Cambridge University Press, 2011.

[160] Xie, Min, Lakshmanan, Laks V. S., and Wood, Peter T. Breaking out of the box of recommendations: from items to packages. In *Proceedings of the 2010 ACM Conference on Recommender Systems, RecSys 2010, Barcelona, Spain, September 26-30, 2010* (2010), pp. 151–158.

[161] Xie, Min, Lakshmanan, Laks V. S., and Wood, Peter T. Composite recommendations: from items to packages. *Frontiers of Computer Science 6*, 3 (2012), 264–277.

[162] Xie, Min, Lakshmanan, Laks V. S., and Wood, Peter T. Generating top-k packages via preference elicitation. *PVLDB 7*, 14 (2014), 1941–1952.

[163] Xie, Min, Lakshmanan, Laks VS, and Wood, Peter T. IPS: an interactive package configuration system for trip planning. *Proceedings of the VLDB Endowment 6*, 12 (2013), 1362–1365.

[164] Xu, Songhua, Jiang, Hao, and Lau, Francis C. M. User-oriented document summarization through vision-based eye-tracking. In *IUI* (2009), pp. 7–16.

[165] Yahoo! finance. http://finance.yahoo.com/.

[166] Yoon, Sung Wook, Fern, Alan, and Givan, Robert. FF-Replan: A baseline for probabilistic planning. In *International Conference on Automated Planning and Scheduling* (2007), vol. 7, pp. 352–359.

[167] Yu, Huizhen, and Bertsekas, Dimitri P. Basis function adaptation methods for cost approximation in MDP. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning* (2009), IEEE, pp. 74–81.