Leveraging Usage History to Enhance Database Usability

Nodira Khoussainova

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2012

Reading Committee:

Magdalena Balazinska, Chair

Dan Suciu, Chair

Bill Howe

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Leveraging Usage History to Enhance Database Usability

Nodira Khoussainova

Co-Chairs of the Supervisory Committee:
Assistant Professor Magdalena Balazinska
Department of Computer Science and Engineering

Professor Dan Suciu
Department of Computer Science and Engineering

More so than ever before, large datasets are being collected and analyzed throughout a variety of disciplines. Examples include social networking data, software logs, scientific data, web clickstreams, sensor network data, and more. As such, there are a wide range of users interacting with these large datasets, ranging from scientists, to data analysts, to sociologists, to market researchers. These users are experts in their domain and understand their data extensively, but are not database experts.

Database systems are scalable and efficient, but are notoriously difficult to use. In this work, we aim to address this challenge, by leveraging usage history. From usage history, we can extract knowledge about the multitude of users' experiences with the database. Consequently, this knowledge allows us to build smarter systems that better cater to the users' needs.

We address different aspects of the database usability problem and develop three complementary systems. First, we aim to ease the query formulation process. We build the SnipSuggest system, which is an autocompletion tool for SQL queries. It provides on-the-go, context-aware assistance in the query composition process. The second challenge we address is that of query debugging. Query debugging is a painful process in part because executing queries directly over a large database is slow while manually creating small test databases is burdensome to users. We present the second contribution of this dissertation:

SIQ (Sample-based Interactive Querying). SIQ is a system for automatically selecting a good small sample of the underlying input database to allow queries to execute in realtime, thus supporting interactive query debugging. Third, once a user has successfully constructed the right query, they must execute it. However, executing and understanding the performance of a query on a large-scale, parallel database system can be difficult even for experts. Our third contribution, PerfXplain, is a tool for explaining the performance of a MapReduce job running on a shared-nothing cluster. Namely, it aims to answer the question of why one job was slower than another. PerfXplain analyzes the MapReduce log files from past runs to better understand the correlation between different properties of pairs of job and their relative runtimes.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisors Dan Suciu and Magdalena Balazinska. I could not have asked for a better advisory team. Magda and Dan are encouraging, incredibly smart, and fun to work with. I owe my research career to them. It is clear how much they care about their work from the way they view every challenge as an exciting opportunity, and the way their eyes light up when learning about a neat algorithm or concept. This passion is simply contagious, and it was inevitable that I, too, would catch the big data fever.

However, being a great advisor is not only about being passionate about the research. It requires dedication and patience, which Dan and Magda had plenty of. They spent countless hours discussing ideas, problems, and algorithms with me, replied to many emails sent after three in the morning, pushed me and told me off when I needed it, helped me create and perfect presentations, edited and re-edited many paragraphs of text, and worked with me on many different projects until I found something that I was excited about. Indeed, we worked on projects ranging from data cleaning, to privacy, to event extraction, to annotations, to collaborative query management, which eventually evolved into the core of this thesis: database usability.

In addition to my official advisors, I would also like to express my gratitude for my pseudo-advisors. Bill Howe has been like a third advisor in the past few years. We have worked on interesting projects together, and have had great conversations about e-science, research, and career paths in general. James Fogarty helped us design user studies for one of the predecessor projects to this thesis work, and I have always appreciated the great questions he asks in my talks; especially the questions that database researchers forget to ask.

Now we go back in time a bit. I would like to thank Phil Bernstein for responding to an email back in late 2004 from an unknown girl in New Zealand, which lead to an exciting research experience at Microsoft Research. Even prior to that, thank you to Gill Dobbie and John Hosking, who gave me my first research opportunities at the University of Auckland, and have been consistently supportive since then. I am also thankful for the North Habour Club in Auckland, not only for the awards, but also for the interview process which helped me formulate some of my career goals early in life. Thank you especially to the late Ross Finlayson, with whom I only interacted a handful of times, yet whose character and work have left a lasting impression on me.

I owe a lot of my excitement about database usability to Alon Halevy, with whom I had the opportunity to do two internships at Google. Even though Alon left UW before I arrived, his support has been felt throughout my graduate school career. From the graduate school decision process, to advice about doing start-ups, all the way through to the job search, Alon has been there to chat, to help, and to share great stories.

Lindsay Michimoto is officially the graduate program advisor, but is really like a mother to all students in the department. I have always appreciated her calming presence, her kind words, and her oracle knowledge of the end-to-end process for acquiring a PhD. It is impressive how nice she has been despite the fact that our first interaction involved me trying to aggressively negotiate my graduate student salary.

I have been extremely fortunate to have been a part of the database research group at UW. First, I would like to thank those who were older graduate students back when I was a young student, including but not limited to Mike Cafarella, Chris Ré, Luna Dong, Julie Letchner and Evan Welbourne. They set the gold standard, provided mentoring, and showed us that the generals exam is a challenge that can indeed be overcome. Similarly, the two postdocs in our group, Wolfgang Gatterbauer and Alexandra Meliou have also been great role models. I owe a special thank you to YongChul Kwon, who I have worked with closely on several occasions. However, I especially want to thank him for the times when I was not working with him, and he still helped solve countless problems that I could not solve on my own for days on end. Thank you also to the rest of the database group, including the undergraduate students that I have worked with, especially Wei-Ting Liao and Leilani Battle who were involved in this thesis work. I have enjoyed the many dinners, seminars, brainstorming meetings, hacking sessions, and the adventures in foreign cities.

Though research is rewarding, life would not be complete without a good set of friends. It is impossible to list everyone and not forget someone. So to take control, I have intentionally left out someone who is very important to me. You know who you are. (This technique is borrowed from Halperin [67]). First, thank you to never-grads, especially to John John, Kayur Patel, Paul Pham, Suporn Pongnumkul, and Elizabeth Tseng. I have enjoyed every one of our adventures together, starting all the way from our outing to Six Flags (aka three flags), our algorithms all-nighters, and the birthday trip to Vancouver. Second, though I do not know the name of your group, thank you to the year following ours. You have provided me with great friends and roommates, including Saleema Amershi, Ben Birnbaum, Daniel Halperin, and Alexander Jaffe. I also appreciate the friendships with those who came before us, including Neva Cherniavsky, Mira Dontcheva and Ian Simon, for showing me how to survive through grad school, and for the younger graduate students for giving me a chance to pass on the wisdom: Abe Friesen, Chloé Kiddon, and Franzi Roesner. Finally, let's not forget the non-students. Thank you to Emily Jacobson, James Lee, Ian Mcdonald, and Anup Rao, for putting up with my graduate student lifestyle. Thank you friends for the laughs, the tears, the terrible practice talks you survived through, the girls nights, the road trips with endless podcasts, the dancing, the drinking, the cooking, the eating, the dust storms, the Vegas trips, and just all the memories.

I would like to finish with family. Thank you to my parents for showing me the world, and for leaving behind everyone and everything you knew to give us better opportunities. You have taught me too many things about life to enumerate here. My older brother Bekh who has always been there for me, and who I feel increasingly closer to as we get older, even though we have lived almost across the world from each other for a large portion of our lives now. My family has been extremely supportive, understood the importance of graduate school, while still reminding me that there is more to life than just school. I will also be forever grateful to my American grandparents, Ray and Elaine Meuse, who not only welcomed me into their home for a whole year, but have always treated me like a daughter. They did the same for my father more than a decade earlier. Words can not express how grateful and appreciative I am for Alexander Jaffe, who has stood by my side for this chapter of my life, and hopefully for many chapters to come. I am so happy that you have been through this whirlwind adventure with me; all the way from watching my quals practice talk in which I ran out of the room due to frustration at my talk quality, to surprising me at my last talk before I become a doctor. I would also like to thank Kether Axelrod and Robert Jaffe for accepting me as family as soon as you met me back around Thanksgiving 2006, for the hilarious email cards that always put a smile on my face, and for creating a wonderful son who also always puts a smile on my face.

DEDICATION

I dedicate this thesis to my grandfathers. One for the piggy-back rides he gave as rewards for solving math problems, and one for teaching me to live life with passion. Both for giving me two wonderful parents who have given me the grounding to do anything I set my mind to.

Chapter 1
INTRODUCTION

Large datasets are being collected and analyzed throughout a variety of disciplines. For example, sociologists analyze social networking data to study relationships and how interactions evolve over time [3, 14, 89], programmers inspect software logs to identify software bugs that may have otherwise been overlooked [128], astronomers run large-scale cosmological simulations to explore how structure evolves in the universe over billions of light years [91], web experts scrutinize clickstreams to understand online behavior and thus provide more personalized services [52], and companies survey their sensor data that tracks inventory to optimize their operations [61]. The scale of many datasets being analyzed is impressive. For example, the Earth Microbiome Project [12] expects to produce 2.4 petabases in their metagenomics effort. As another example, the Venture Development Corporation [16] predicts that Walmart will generate around seven terabytes of data every day when RFID tags are used for each item. The upcoming Large Synoptic Survey Telescope (LSST) [6] is estimated to generate fifteen terabytes of raw data per night, for a total of five petabytes per year. Meanwhile, social networks such as Facebook [3] and Twitter [14] are also collecting extensive amounts of data. For example, at the end of the month of March 2012, Facebook had a total of 901 million active users, and 125 billion friend connections, and Twitter had an average of over 140 millions 'tweets' sent per day (as of March 2011) [15].

Traditional database management systems (DBMS) [111, 99, 121, 78], and modern day large-scale, parallel data analytic systems [45, 123] are designed and built for organizing, storing, managing, and analyzing large datasets. Over the past 40 years, database systems have evolved to become scalable and efficient. However, they remain notoriously difficult to use [83]. Furthermore, as the popularity of large scale, parallel database systems for massive data analytics increases, there will be increased demand from users for help in using these systems without the support of a database administrator or some other form of expert team.

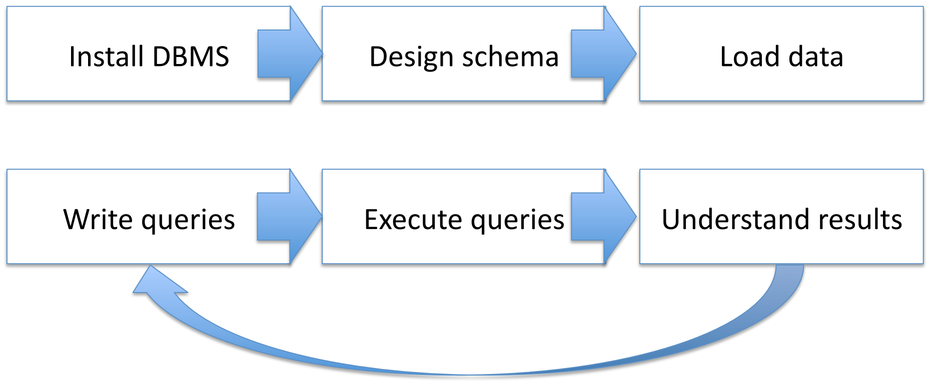


Figure 1.1: Challenges in the data analysis process.

We outline some of the challenges in the data analysis process in Figure 1.1. We partition the challenges into two types: the *one-off challenges* that a user faces when trying to start using a database system, and the *repeated challenges* that a user faces every time he or she wants to ask a question over the data. Though both sets of challenges are interesting and important, we only address the repeated challenges in this thesis work. We still describe both sets of challenges, and investigate the related literature for both.

The first set of challenges (as pictured in the top half of Figure 1.1) are those faced at setup time, including but not limited to installing a DBMS, designing a schema, and loading the data into the database. These tasks are difficult for various reasons. For example, schema design is difficult as it requires an advanced understanding of the relational model, normal forms, database optimization, as well as a good understanding of the data itself before the user is able to explore or analyze the data (since the schema must be created before the data is loaded into the DBMS). Even though these challenges are faced only once, they are important to address because they form the initial roadblock to database technology adoption. There is a concerted effort in the database community to ease these initial pains. These include projects such as SQLShare [73] that allows users to upload data files using a web browser (thus avoiding the need to install a DBMS) and execute queries over the data immediately (before having to design a schema), as well as work on schema-later approaches [66, 112]. Advances in industry are also eliminating some of these one-off challenges. For example, cloud providers such as Amazon [2], VMWare [17], Salesforce.com [9] are offering Platform as a Service, thus saving users from having to install their own DBMS.

The second set of challenges, the repeated challenges, are those that the user must face every time he or she wants to answer a question over the data. The first challenge is in formulating the query. This is often difficult because SQL is a difficult language to grasp at first, especially for users who are familiar with imperative programming languages since SQL is a declarative one. Second, there are many cases where users write queries over large databases that they neither created nor populated, and thus unfamiliarity with the schema can make queries difficult to write. Even once the query is formulated, the difficulty continues. The debugging of queries is not interactive. The feedback loop can be slow if users execute their partially correct queries over the entire, large-scale database. Furthermore, as more data analysis moves to MapReduce-type systems, executing and optimizing queries in such environments requires the user to perform all the tuning, as opposed to traditional DBMSs, which provide automatic tuning and optimization. Finally, even after executing the query, often understanding the results of the query also requires a lot of work because the output itself can be a large dataset.

There is a diverse body of recent work on addressing some of the repeated challenges in database usage. First, to ease the difficulty of writing SQL queries, many projects have proposed different querying methods ranging from visual querying, querying by forms, and keyword-based querying, as well as techniques for summarizing schemas to help users who are unfamiliar with the schema [27, 35, 76, 101, 85, 26, 29, 74, 105, 104, 106, 131]. Second, though it is not yet as advanced as in traditional DBMSs [28, 43, 21, 20, 39, 7, 77, 8, 13], there is significant work on automatic performance tuning in large-scale, parallel data analytic systems [24, 69, 59, 57, 51, 122]. Third, to help users more easily process and understand the results of their queries, several efforts are directed toward supporting easier visualization of data [120, 5] and ranking output tuples [18].

While these projects ease some of the pain of query construction, debugging, and execution, in this dissertation, we take a radically different approach. We propose to leverage an increasing trend: collaboration. Namely, an increasingly common database usage pat-

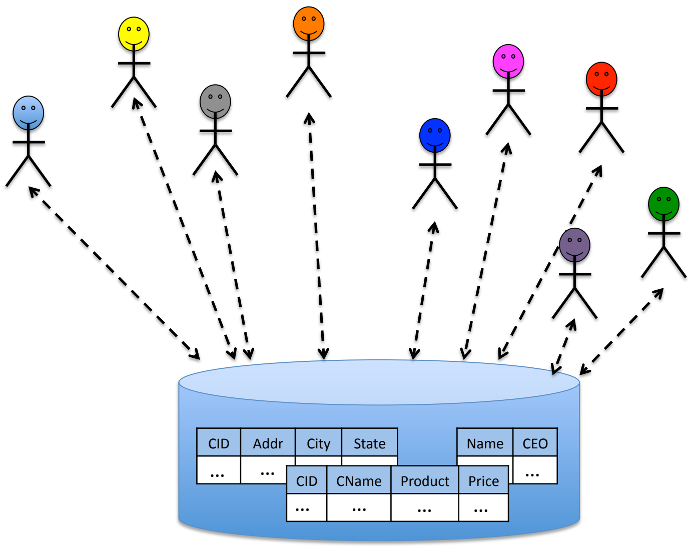


Figure 1.2: Collaborative environments where users perform their own data analysis tasks over a shared database.

tern is where some party creates a large database and makes it available to a community of users. Then, these users all perform their own data analysis and exploration over this shared database (as depicted in Figure 1.2). For example, the Sloan Digital Sky Survey (SDSS) is a famous example of this shift toward collaborative, data-intensive analytics [119]. SDSS has mapped 25% of the sky, collecting over 30 TB of data (images and catalogs), on about 350 million celestial objects. SDSS has had a transformative effect in astronomy not only due to the value of its data, but because it made that data accessible online through web forms and SQL queries. To date, astronomers and others have submitted over 20.7 million SQL queries to the SDSS database. In such settings, if we could utilize the efforts of all the users of the shared database, we would be able to leverage the collective knowledge and experience of all past users to help ease the pains of new interactions with the database.

This hypothesis forms the primary focus of this dissertation work. The question is whether utilizing usage history can indeed speed-up and improve new interactions with the data. We investigate this question in three different contexts, all related to the usability challenges in the data analysis lifecycle. Namely, in the query composition, query debugging, and performance debugging components. We show that via storing, querying and mining past usage logs, we can extract knowledge about the multitude of users' experiences with the database. Consequently, this knowledge allows us to build smarter systems that cater better to the users' needs.

In this thesis work, we investigate three different problems and explore how we can solve the problems by utilizing database usage history. We build a research system to prototype our solution for each problem. In the first project, we aim to ease the query formulation process. We build the SnipSuggest system [88], which is an autocompletion tool for SQL queries. It provides on-the-go, context-aware assistance in the query composition process. SnipSuggest recommends possible additions to various clauses in the query using relevant 'snippets' collected from a log of past queries. The second challenge we address is that of query debugging. Query debugging is a painful process in part because executing queries directly over a large database is slow while manually creating small test databases is burdensome to users. The second contribution of this thesis is SIQ (Sample-based Interactive Querying). SIQ is a system for automatically selecting a 'good' small sample of the underlying input database to allow queries to execute in realtime, thus supporting interactive query debugging. Third, once a user has successfully constructed the right query, they must execute it. Though this step is fairly simple in a traditional relational DBMS (e.g., the user needs to click on the "play" button), executing and understanding the performance of a query on a large-scale, parallel database system is difficult even for experts. Our third contribution, PerfXplain [87], is a tool for explaining the performance of a MapReduce job running on a shared-nothing cluster. Namely, it aims to answer the question of why one job was slower than another. PerfXplain analyzes the MapReduce log files from past runs to better understand the correlation between different properties of pairs of job and their relative runtimes.

We now outline the goals, challenges, and contributions of SnipSuggest, SIQ, and PerfXplain in more detail.

1.1 SnipSuggest

In the first project, we address the problem of helping the increasing population of non-expert database users, who need to perform complex analysis on their large-scale datasets, but have difficulty writing database queries. We build the SnipSuggest system, which is an autocompletion tool for SQL queries. It provides on-the-go, context-aware assistance in the query composition process.

As a user types a query, SnipSuggest recommends possible additions to various clauses in the query using relevant snippets collected from a log of past queries. SnipSuggest's current capabilities include suggesting tables, views, and table-valued functions in the `FROM` clause, columns in the `SELECT` clause, predicates in the `WHERE` clause, columns in the `GROUP BY` clause, aggregates, and some support for sub-queries. SnipSuggest adjusts its recommendations according to the context: as the user writes more of the query, it is able to provide more accurate suggestions.

We evaluate SnipSuggest over two query logs: one from an undergraduate database class and another from the Sloan Digital Sky Survey database. We show that SnipSuggest is able to recommend useful snippets with up to 93.7% average precision, at interactive speed. We also show that SnipSuggest outperforms naïve approaches, such as recommending popular snippets.

1.2 SIQ

In this project, we address the problem of query debugging. Query debugging is currently a painful process for users largely because executing queries directly over a large dataset is slow. Some users choose to construct a sample dataset, often referred to as a toy database, with which they interact through the debugging process before executing the query over the full dataset. However, constructing a good toy database is difficult and cumbersome.

As the second contribution of this dissertation, we present SIQ (Sample-based Interactive Querying). SIQ is a system for automatically constructing a 'good' small toy database in order to support an interactive query debugging process. The toy database allows all in-progress queries to execute in realtime, and allows the user to quickly understand the effect of the modifications that he or she makes to the query.

SIQ leverages a log of past query sessions to generate a toy database that is small, and illustrative of these past sessions. We evaluated SIQ over a log consisting of queries written for an undergraduate database class homework. We find that SIQ is able to generate a good toy database for a query session without knowing the queries in the query session a priori. Our results also show that the SIQ system generates toy databases that are up to 85 times smaller than a union-based technique that generates a sample per past query and takes the union of all these samples.

1.3 PerfXplain

Once a user has successfully constructed the right query, they must execute it. Though this step is straightforward in a traditional relational DBMS (e.g., the user needs only to click the "play" button), executing and understanding the performance of a query on a large-scale, parallel database system is difficult even for experts.

As the third component of this thesis, we present PerfXplain, a system that enables users to ask questions about the relative performances (i.e., runtimes) of pairs of MapReduce jobs. PerfXplain provides a new query language for articulating performance queries and an algorithm for generating explanations from a log of past MapReduce job executions

We make several contributions in this PerfXplain work. First, we propose a simple language, PXQL, for articulating queries about the performance of a pair of MapReduce jobs. We formally define the notion of an explanation and three metrics relevance, precision, and generality to assess the quality of an explanation. We develop an approach for efficiently extracting explanations that have high relevance, high precision, and good generality from a log of past MapReduce job executions, which is based on techniques related to decision-tree building. Finally, we evaluate the PerfXplain approach on a log of MapReduce jobs executed on Amazon EC2. We show that PerfXplain is able to generate explanations with higher precision than two naïve explanation-generation techniques, and offer a better trade-off between precision and generality.

1.4 Dissertation Overview

The dissertation is organized as follows. In Chapter 2 we discuss the different aspects of databases that make them difficult to use, and summarize the recent efforts for addressing each problem. Then, we present the three projects in chronological order. In Chapter 3 we describe SnipSuggest, a tool for SQL autocompletion. In Chapter 4 we present the

PerfXplain project, whose aim is to explain MapReduce job performance. Chapter 5 outlines the SIQ system, which allows users to perform interactive querying based on samples. We discuss future directions in Chapter 6, and conclude in Chapter 7 with a summary of our contributions.

Chapter 2
RELATED WORK

In this chapter, we describe the existing research and tools that address some typical challenges faced by users when interacting with database systems. We structure the chapter as follows. In Section 2.1, we begin by summarizing a seminal paper in database usability by Jagadish *et al.* [83], which, like this chapter, describes some of the key pain points in database usability. Following this overview, we then dedicate one section for each of the areas highlighted in Figure 1.1. We wrap up the chapter by describing a range of work in other areas, where the underlying technique leverages past usage history.

2.1 The Pain Points of Database Usage

Jagadish *et al.* [83] provide a comprehensive summary of why databases are difficult to use. In this paper, the authors discuss five major pain points of using database systems today, and propose ways in which to address them.

The first challenge is that current, structured query models are powerful but expect users to fully understand the schema of the database. The authors name this challenge 'painful relations'. Schema familiarity can be a high expectation to meet because different users have different ideas of how information should be structured, and thus find the given schema difficult to understand or remember. Furthermore, due to normalization, most schemas that start with cohesive tables have them broken into multiple tables that are often necessary but cumbersome to stitch together. The authors suggest that though a logical schema provides an abstraction over the physical schema of a database, it is still not at the right level of abstraction for users, and that there should be yet another level of abstraction, which the authors call the 'presentation data model'.

Some of the techniques we describe in Section 2.5, for helping in query composition, address the painful relations challenge. For example, the work on schema summarization helps users that are unfamiliar with the database schema.

The second problem explored in the paper is that database systems often overwhelm users with too many options. The authors, fittingly, name this the ‘painful options’ problem. In SQL, the same query can be formulated in multiple ways. If a user does not understand the difference between the options, this leads to confusion and mistrust in the system. This point is exemplified by the fact that form-based querying interfaces, which inherently restrict the space of queries that the user can pose, are extremely popular. On the other hand, forms are often curated by developers, and this can be a tedious process. Furthermore, they are not well-suited for end-users who wish to have flexibility in the queries they can write. In Section 2.5 we describe work on techniques that address some of these shortcomings of form-based interfaces.

The third problem discussed by Jagadish *et al.* is called ‘unexpected pain’, and it explains that too often after a query has executed, the user is unable to understand why a tuple is included in or excluded from the output. The authors propose that database systems be able to explain why or why not a tuple exists or does not exist in the output of a query. The research area related to provenance, which we discuss in Section 2.8, is a good candidate for addressing this pain point.

The fourth pain point discussed is that the current querying procedure often involves the user spending time constructing the query, followed by a significant time waiting for the query to execute, and finally seeing the output only to find that the query was wrong. This long feedback loop problem is called ‘unseen pain’. Most users prefer to directly see and directly manipulate the data. The problem is that current database systems do not provide a what-you-see-is-what-you-get (WYSIWIG) form of interaction. We discuss query techniques that are more WYSIWIG in Section 2.5, tools for predicting the execution time of a query in Section 2.6, and query debugging techniques that shorten the feedback loop in Section 2.8.

Finally, the fifth problem is called the ‘birthing pain’, and refers to the pain of schema design for creating new databases and the burden of learning the schema before being able to add data to it. We devote Section 2.3 to exploring the challenge of schema design and the research that eases this pain.

2.2 DBMS Installation

Before a user can explore and analyze a dataset, he or she must first install and configure a database system. Though this is a challenge with traditional DBMS's, this step is mostly bypassed by many cloud providers that offer Platform as a Service. Examples include Amazon's AWS service [2], VMWare [17], and Salesforce.com [9].

The SQLShare [11] project also removes the DBMS installation obstacle for users. It is a tool, built over a cloud platform, for scientists that allows its users to upload their data, write queries over it, and share the data. It actually addresses all three of the one-off challenges as outlined in Figure 1.1 (i.e., DBMS installation, schema design and data loading) by providing web browser access (i.e., no installation required), supporting data upload in a single button click, and allowing users to execute queries on their data immediately after uploading it without having to first design a schema.

2.3 Schema design

Next, the user must design a schema for their data. Schema design is a difficult process, requiring a good understanding of the relational model, normal forms, database optimization, and more. It is unreasonable to expect knowledge of any of the above from a non-expert database user. Furthermore, the user is also expected to design the schema prior to being able to explore or analyze the data (i.e., the schema must be designed and created before the data is loaded).

The schema-later approach [36, 66, 112] and the pay-as-you-go paradigm from the DataSpaces work [66], are a good step toward easing this challenge. In this work, the authors present a new abstraction called a DataSpace, which is a collection of data sources and relationships between them. The goal of a DataSpace is to accept all data, regardless of its format, location or whether it has a schema, and provide best-effort services on top of that data. The pay-as-you-go paradigm refers to how it requires no up-front cost to use the platform. Additionally, the DataSpace learns from users' interaction with the data, to discover semantic relationships between data items and perhaps even infer schemas, as well as identify and streamline common information tasks.

The CRIUS [112] project also provides a schema-later approach. CRIUS is a system that allows end-users to create and modify their schema on-the-go, as new data arrives that does not fit the current schema. The system supports nested data. The CRIUS system allows schema modifications in the form of dragging and dropping column headers. It is able to support all the schema modifications that the nested relational algebra supports. Such an approach allows the user to start with a simple schema, and evolve it one-step-at-a-time as the need arises. Additionally, CRIUS extracts and incrementally maintains a set of functional dependencies, that are satisfied by the existing data. This allows CRIUS to help in two ways. First, it can auto-complete parts of new entries. Second, it can prevent errors. For example, suppose the current table contains a row `(Alice, 4259843545, 1234 Crius Ave)` and one of the functional dependencies inferred is $name \rightarrow phone$. If the user were to insert a new row, and had typed `Alice` into the *name* column, then CRIUS would automatically fill in the *phone* column with `4259843545`. Now consider if the user modifies Alice's phone number in only one of these rows. At this point, CRIUS asks the user whether she would like to update the other Alice row, or to force the update, thus invalidating the inferred $name \rightarrow phone$ functional dependency. In summary, CRIUS allows end users to rapidly design and easily refine schemas, thus largely eliminating the challenge of schema design.

2.4 Loading data

Loading data into a database is known to be a painful task, despite its conceptual simplicity. First, the data must be transformed into files that adhere to the DBMS's strict formatting restrictions. Second, copying the data into the database can take a long time, especially if the data is not bulk-loaded in.

Several commercial DBMS's now support "external" tables, allowing users to query flat files without loading them into the database. They have not yet adapted optimization algorithms for external tables, thus leading to slow query processing on these tables. A recent paper [80] argues for new DBMSs which better support these "external" tables, thus solving the problems of when and how to load different parts of the data, how to store the data that has been loaded, as well as how to access each data part. The paper is a vision paper, and thus does not provide any algorithms to solve the challenges. However, they do investigate a few simple techniques, including adaptive loading (where the system stops parsing a row, as soon as it has extracted the required columns from it, or determined that the row does not pass a predicate), dynamic file partitioning, and more.

As discussed in Section 2.2, SQLShare also addresses the data loading problem by providing a simple mechanism for loading data through a web interface.

2.5 Query Composition

Once the data is in the database, users can finally pose queries over it. We overview projects that aim to ease the SQL composition process.

Visual query tools. One reason why SQL is difficult to write is because it is a non-visual programming language. To address this problem, several systems offer visual querying [27, 35, 76, 101]. Query-by-example [133] is also an example of a visual querying language. Furthermore, there is a large body of work on visual mashup constructors, including Google Mashup [62], Popfly [100], and Yahoo! Pipes [129]. A mashup is an application that combines data from multiple sources to create a new service. Though a mashup is not exactly a SQL query, some of the techniques could potentially be applied to constructing SQL queries. More recent work presents AppForge [130], a system for building applications in a WYSIWYG environment. It extends the mashup editors above by supporting stateful applications (where the developer can save state to a backend database), as well as by supporting a more closely integrated, WYSIWYG development and execution environment.

Visualizations. A natural method for making sense of a large dataset is to visualize it. In a sense, creating a visualization, is equivalent to writing a query over the data. In fact, it is often the case that the visualization system represents the visualization as a SQL query in the background. There are thousands of research projects and commercial products, which focus on visualizing different kinds of data, and hence we do not attempt to summarize the area here. As an example though, Tableau [120] allows users to interactively and rapidly construct visualizations, and hence the underlying query via a graphical interface. Google Fusion Tables [5] also provides this functionality.

Other projects that generate SQL queries to describe visualizations include work by

Heer *et al.* [68], the DEVise system [94], the Visual Query Environment work [46], and Improvise [126]. In these projects, the user graphically selects a set of items, and the system generates a matching SQL query. For example, Heer *et al.* [68] use this translation from visualization into SQL, followed by a relaxation of the query in order to allow the user to interactively generalize their selection.

Forms. Because SQL is difficult to write, many applications provide a form-based interface for accessing databases. However, these forms are usually built by application developers and not the end-users. Additionally, they are usually very restrictive in the type of queries they support. Thus, they are not well-suited for users who want to execute ad-hoc queries that evolve frequently. To address this problem, Jayapandian and Jagadish [85] present an algorithm for automatically constructing query forms given a database schema and content. The algorithm proceeds by first selecting the "useful" schema elements (based on the schema and data), and then constructing forms from these elements.

Keyword-based querying. Search engines have provided a simple interface for querying the web via keyword search. Their usability is a key factor contributing to their widespread success. Several research projects aim to provide keyword-based querying capabilities over relational databases [26, 29, 74, 105, 104, 106] and XML databases [64].

Nandi *et al.* [105] present a technique for phrase autocompletion for keyword search over structured databases. A similar tool [104] allows users to construct search queries without knowledge of the underlying schema. As the user types in the search box, the tool starts suggesting elements of the schema, followed by fragments of text from the database content. This tool supports standard conjunctive attribute-value queries.

In the Qunits project [106], Nandi and Jagadish introduce the notion of a 'qunit', which is a 'basic, independent semantic unit of information in a database'. Qunits are not constrained to the schema of the database or the relational model. Instead, they fit the user's mental model of how the database is organized. For example, if the user views the Internet Movie Database (IMDb) as a collection of movies and their casts, then each qunit can represent this concept (ignoring the underlying schema which breaks the database into a Movies table, an Actors table, and a Casts table). Using qunits, the database is now modeled as a heterogeneous collection of independent qunits. Consequently, standard information retrieval techniques, that typically run over collections of heterogenous documents, can be applied.

Li *et al.* [93] address the problems of keyword search, form-based search, and SQL-based search usability in a single system they call DBease. It provides a search-as-you-type capability, through the use of Trie structures. In addition, to support SQL, as a user types in keywords, DBease recommends relevant SQL queries on-the-fly. At this point, the user can select a specific SQL query and execute it.

Query by output. Sometimes it is easier to describe the output of a query, rather than writing the query itself. Given the output of a query and the underlying database, the Query-by-Output project [125] is able to automatically construct a query that produces the specified output. The paper considers the case where the original query is known, in which case it generates an 'instance-equivalent' query, as well as the case where the original query is not known. In the context of composing queries, the techniques from the latter case could be used to design a new form of querying where the user specifies the approximate output that she desires, and the system automatically constructs the query to produce this output.

SQL to English. A common way that non-expert database users formulate queries is through copying and pasting queries written by other users or even sample queries found on the web. The user starts with the existing query and modifies it until they reach their desired query. The challenge here is that sometimes SQL queries are difficult to understand. Annotating a query with a description can increase its understandability, but users can not be expected to annotate every query they write. Koutrika *et al.* [90] present methods for automating this process. Namely, they present an algorithm for translating SQL into natural language.

Querying an unfamiliar database. There are many examples of large databases that domain experts interact with daily. In the sciences, for example, consider the Sloan Digital Sky Survey (SDSS), a 30 TB database which consists of 88 tables, 51 views, 204 user-defined functions, and 3440 columns! Other examples include the Incorporated Research Institutions for Seismology (IRIS) [82], and soon the Large Synoptic Survey Telescope (LSST) [6]. To give an idea for the scale of these projects, the LSST is estimated to generate fifteen terabytes of raw data per night [6] for a total of five petabytes per year.

In the above cases, the users are writing queries over databases that they did not create, and therefore may be unfamiliar with. Combined with the difficulty of writing SQL, unfamiliarity with the underlying database makes the query formulation process more difficult, and even insurmountable at times.

Two projects address this problem by providing summarizations of the underlying schema. Yang *et al.* [131] automatically generate a summary of a large database based on its schema and data content. In this work, summarization involves identifying the most important tables, and then clustering all the tables into groups of related tables. In [132], the authors also perform schema summarization, but in the context of XML databases.

Another paper [37] addresses this problem by recommending SQL queries for database exploration. QueRIE [37] analyzes a user's query log, finds other users who have executed queries over similar parts of the database, and recommends new queries to retrieve relevant data. This work helps users who have had significant interactions with the database, but may not yet be familiar with other portions of the schema.

2.6 Query Optimization and Execution

After formulating the query, sometimes the user will need to perform tuning to ensure the efficient execution of the query. Though many traditional DBMS's provide automatic tuning and optimization, in many of the new parallel data-analytic services, the user still needs to perform all the tuning herself.

Database performance tuning. Many existing database management systems provide tools to examine and tune the performance of SQL queries including Teradata [28], Oracle [43], SQL Server [21, 20, 39], MySQL [7], DB2 [77], Postgres [8, 13], and others. These tools focus primarily on tuning the physical and logical designs of a database, and are aimed at database administrators.

More recent work has focused on MapReduce tuning and optimizations [69, 47, 86, 116, 84]. Many of these techniques address the problem from a different perspective from traditional database optimizers. Consider Xplus [69] as an example. Given a query, whereas existing database optimizers search for a plan first, and then execute the plan, Xplus tries to find a better plan by running a small set of plans proactively, collecting monitoring data from these runs, and iterating until it finds a better plan.

Autonomic databases. Another line of work strives to make database systems self-configurable, self-tuning, or autonomic, including MapReduce systems [24, 70]. A key project in this space is the Auto-Admin project, which aims to make DBMS's self-administrating [40, 38, 22, 19]. This includes automatically selecting indexes and materialized views [40, 22], and allowing a DBA to explore hypothetical configurations [38]. Research on autonomic databases includes work on automating failure diagnosis [50].

2.7 Understanding Query Results

Queries over large datasets can lead to results consisting of millions of rows, or more. Understanding and extracting knowledge from such large results can be difficult and time-consuming.

Visualizations. As discussed in Section 2.5, visualizations can significantly reduce the time to understand a dataset, including the results of a query. We refer the reader back to Section 2.5 for more details of projects related to visualizations.

Ranking output tuples. An alternative approach to making sense of a query's result is to rank the output tuples. Agrawal *et al.* [18] provide a good summary of how to apply ranking techniques used in information retrieval to ranking the output tuples of a SQL query.

2.8 Query Debugging

Query debugging usually occurs amidst the query composition process, but we address it on its own in this section. Query debugging is especially difficult when the query is over a large dataset, and each execution of the query can take minutes, or hours.

Debugging with sample data. As a user constructs a query, it is often too expensive to execute the query after each modification, because each execution takes a significant amount of time. To address this problem, many users choose to create a sample data set, and test their queries on this dataset. However, constructing such a sample dataset is a difficult and tedious task, because selective operators (such as selective filters or joins) can lead to empty results over the sample. In [107], Olston addresses the problem, specifically for debugging Pig [109, 60] programs. Given a Pig program, Olston proposes a novel technique for automatically generating small sample datasets that are illustrative of each operator's semantics.

The problem of synthetic data generation has been studied for decades [96, 32, 23, 30, 95]. Besides Olston's work, the goals of this line of work vary from generating data to test new database system components, database application testing, and benchmarking.

Mannila et al [96] study the problem of query-aware data generation. In this work, the goal is to generate a small, test database that is illustrative and complete for a Select-Join-Project (SPJ) query. A test database is illustrative if every operation has a direct effect on the output, and is complete if it distinguishes the query at hand from all other queries in the language of queries (SPJ queries in this case).

A seminal paper in this area is by Bruno and Chaudhuri [32], and it introduces a general language called the Data Generation Language (DGL) for specifying distributions for synthetic databases. For example, using DGL, one can specify that the number of lineitems per order follows a Zipfian distribution and that customers make purchases from vendors only in their nation.

Similar to Mannila et al [96], the QAGen [30] and MyBenchmark [95] projects explore the problem of query-aware data generation for DBMS testing. As input, the QAGen project takes a query plan annotated with cardinality constraints on each operator node. As output, it produces a synthetic dataset that satisfies these constraints. Overall, it starts with a symbolic database, which is similar to a traditional databases except that instead of constants, they have variables as values. Using symbolic query processing, the algorithm then applies the constraints from the query plan to the symbolic database. Finally, the authors use an out-of-the-box constraint satisfaction program (CSP) to instantiate the symbolic database with real values. The MyBenchmark project extends the algorithm to generate synthetic databases for sets of query plans. The input to MyBenchmark is a set of annotated query plans, and the output is a set of database instances that satisfies the annotation constraints.

Arasu *et al.* [23] work on a similar problem as the MyBenchmark project, except that as output they aim to generate a single database. Additionally, as input, Arasu *et al.* take a set of queries together with cardinality constraints. A cardinality constraint specifies that the output of a query over the generated dataset must have a specific cardinality. They show that cardinality constraints can be expressed using annotated query plans and vice versa.

Provenance. When debugging a SQL query, users often ask “why is this tuple in my output?” and “why is this tuple *not* in my output?”. To answer these questions, recent research efforts propose storing, maintaining, and querying the provenance of output tuples.

Several projects focus on defining and modeling provenance [33, 41]. In general, provenance is split into three types of provenance: how, why, and where provenance. Green *et al.* [63] find a model for representing many types of provenance through the use of semirings. Recent work has focused on explaining why a certain tuple is not in the output. Some work presents the provenance of non-answers [75]. Another project, Artemis [71], focuses on how to tweak the data (via insertions, edits, or deletions) to yield the missing tuples. Tran *et al.* [124] investigate how to tweak the query to yield the missing tuples. Meliou *et al.* [97, 98] propose causality as a unified framework for explaining both the answers and non-answers of queries.

2.9 Utilizing Usage History

So far, we have described various projects related to database usability. We now discuss a potpourri of projects related to reusing the past efforts of users.

Many projects aim to collect and reuse user actions, usually in a collaborative setting. For example, CoScripter [42] is a web-based system for recording, automating and sharing web browser processes. Users can automate and share tasks such as printing photos online, checking flight departure times, etc. Also, many web mashup editors [62, 79, 100, 129] enable users to construct and store mashups of data from different sources on the web, and later share these mashups with the public.

In the scientific realm, there are many systems for managing scientific workflows, with the goal of saving the time and effort of users through reuse. For example, the MyExperiment project [115] focuses on community oriented sharing and common workflow platform for many science and engineering disciplines while Scriptome [118] focuses on sharing well-curated script templates for computational biologists. The VisTrails project is one of the most advanced such workflow management systems. In some recent work, they demonstrate advanced capabilities of the system such as query-by-example and query-by-analogy with visualizations [117].

In addition to only managing past usage history, many projects actually mine usage history. For example, in the web search community, many projects analyze keyword search logs [31, 48, 49, 92], for goals ranging from predicting the next user action to designing a taxonomy of searches.

Chapter 3

SNIPSUGGEST: AN AUTOCOMPLETION SYSTEM FOR SQL

SQL is having a transformative effect on science. Authoring SQL queries, however, remains a challenge for the vast majority of scientists. Scientists are highly-trained professionals, and can easily grasp the basic select-from-where paradigm, but to conduct advanced scientific research, they need to use advanced query features, including group-by's, outer-joins, user defined functions, functions returning tables, or spatial database operators, which are critical in formulating their complex queries. At the same time, they have to cope with complex database schemas. For example the Sloan Digital Sky Survey schema has 88 tables, 51 views, 204 user-defined functions, and 3440 columns [119]. One of the most commonly used views, PhotoPrimary, has 454 attributes! The learning curve to becoming an expert SQL user on a specific scientific database is steep.

As a result, many scientists today leverage database management systems only with the help of computer scientists. Alternatively, they compose their SQL queries by sharing and re-using sample queries. In a small-scale survey that we performed among scientists, we found that 86% of scientists have looked at other users' queries to help them compose their own. The SDSS website provides a selection of 57 sample queries, corresponding to popular questions posed by its users [10]. Similarly, SQLShare provides a "starter kit" of SQL queries, translated from English questions provided by researchers. *Scientists who write complex SQL today do this through cut and paste.* The challenge with sample SQL queries is that users either have access to a small sample, which may not contain the information that they need, or they must search through massive logs of past queries (if available), which can be overwhelming.

Assisting users in formulating complex SQL queries is difficult. Several commercial products include visual query building tools [27, 35, 76, 101], but these are mostly targeted to novice users who struggle with the basic select-from-where paradigm, and are not used by scientists. More recent work [131] has proposed a method for clustering and ranking relations in a complex schema by their importance. This can be used by an automated tool to recommend tables and attributes to SQL users, but it is limited only to the most important tables/attributes. Scientists are experts in their domain, they learn quickly the most important tables. Where they truly need help are precisely the “advanced” features, the rarely-used tables and attributes, the complex domain-specific predicates, etc. Some new systems, such as QueRIE [37], recommend entire queries authored by other users with similar query patterns. These, however, are designed for users who have already written multiple queries, and wish to see past queries that touch a similar set of tuples.

In this project, we take a radically different approach to the problem. We introduce SnipSuggest, a new SQL autocomplete system, which works as follows. As a user types a query, she can ask SnipSuggest for recommendations of what to add to a specific clause of her query. In response, SnipSuggest recommends small *SQL snippets*, such as a list of k relevant predicates for the WHERE clause, k table names for the FROM clause, etc. The key contribution is in computing these recommendations. Instead of simply recommending valid or generally popular tables/attributes, SnipSuggest produces *context-aware suggestions*. That is, SnipSuggest considers the partial query that the user has typed so far, when generating its recommendations. Our key hypothesis is that, as a user articulates an increasingly larger fragment of a query, SnipSuggest has more information on what to recommend. SnipSuggest draws its recommendations from similar *past queries authored by other users*, thus leveraging a growing, shared, body of experience.

This simple idea has dramatic impact in practice. By narrowing down the scope of the recommendation, SnipSuggest is able to suggest rarely-used tables, attributes, user-defined functions, or predicates, which make sense only in the current context of the partially formulated query. In our experimental section, we show an increase in average precision of up to 144% over the state-of-the-art (Figure 3.5(c)), which is recommendation based on popularity.

More specifically, our project makes the following contributions:

1. We conduct a small-scale survey to understand how scientists use DBMSs today. We

study various aspects of their database usage, including how often they write queries, and how they learn from others' queries. We report our findings in Section 3.1.

2. We introduce *query snippets* and the *Workload DAG*, two new abstractions that enable us to formalize the context-aware SQL autocomplete problem. Using these abstractions, we define two metrics for assessing the quality of recommendations: *accuracy* and *coverage* (Section 3.4).

3. We describe two algorithms *SSAccuracy* and *SSCoverage* for recommending query snippets based on a query log, which maximize either accuracy or coverage (Sections 3.4.4, 3.4.5).

4. We devise an approach that effectively distinguishes between potentially high-quality and low-quality queries in a log. We use this technique to trim the query log, which drastically reduces the recommendation time while maintaining and often increasing recommendation quality (Section 3.5).

5. We implement the above ideas in a SnipSuggest prototype and evaluate them on two real datasets (Section 3.6). We find that SnipSuggest makes recommendations with up to 93.7% average precision, and at interactive speeds, achieving a mean response time of 14ms per query.

3.1 Small-scale Survey of Database Usage among Scientists

To better understand how scientists use DBMSs today and, in particular, how they query these databases, we carried out a small-scale, informal, online survey. Our survey included 37 questions, mostly multiple-choice ones and took about 20 minutes to complete. We paid respondents $10 for their time. Seven scientists from three domains responded to our survey (four graduate students, one postdoc, and two research scientists); these scientists have worked with either astronomical, biological, or clinical databases.

Of interest to this chapter, through this survey, we learned the following facts. All respondents had at least one year experience using DBMSs, while some had more than three years. Three scientists took a database course, whereas the other four were self-taught DBMS users. Four participants have been working with the same dataset for over a year, while three of them acquired new datasets in the past six months. The data sets ranged in size from less than one gigabyte (one user), to somewhere between one gigabyte and one terabyte (five users), to over a terabyte (one user). The reported database schemas include 3, 5, 7, 10, 30, and 100 tables. One respondent did not report the number of tables in his/her database. All respondents reported using a relational DBMS (some used other alternatives in addition).

More interestingly, three participants reported writing SQL queries longer than 10 lines, with one user reporting queries of over 100 lines! All users reported experiencing difficulties in authoring SQL queries. Two participants even reported often not knowing which tables held their desired data.

Five respondents reported asking others for assistance in composing SQL queries. All but one reported looking at other users' queries. Five participants reported looking "often" or even "always" at others' queries, whereas all participants either "often" or "always" look for sample queries online. Three participants mentioned sharing their queries on a weekly to monthly basis. Finally, all but one user save their own queries/scripts/programs primarily in text files and reuse them again to write new queries or analyze different data.

The findings of this informal survey thus indicate that many scientists could potentially benefit from tools to share and reuse past queries.

3.2 Motivating example

In this section, we present an overview of the SnipSuggest system through a motivating scenario based on the SDSS query log.

Astronomer Joe wants to write a SQL query to find all the stars of a certain brightness in the r-band within 2 arc minutes (i.e., $\frac{1}{30}$th of 1°) of a known star. The star's coordinates are 145.622 (ra), 0.0346249 (dec). He wants to group the resulting stars by their right ascensions (i.e., longitudes). This is a *real query* that we found in the SDSS query log. Joe is familiar with the domain (i.e., astronomy), but is not familiar with the SDSS schema. He knows a bit of SQL, and is able to write simple select-from-where queries.

It is well known that `PhotoPrimary` is the core SkyServer view holding all the primary survey objects. So, Joe starts off as follows: `SELECT * FROM PhotoPrimary`.

Joe is interested in only those objects that are near his coordinates. Browsing through the 454 attributes of the PhotoPrimary table's schema fails to reveal any useful attributes over which to specify this condition. Joe suspects that he needs to join PhotoPrimary with some other table, but he does not know which one. Joe thus turns to SnipSuggest for help and asks for a recommendation of a table to add to his FROM clause.

SnipSuggest suggests the five most-relevant snippets for this clause: `SpecObj`, `Field`, `fGetNearbyObjEq(?,?,?)`, `fGetObjFromRect(?, ?, ?, ?)`, and `RunQA`. (All the suggestions in this section are real suggestions from SnipSuggest.)

In this example, `fGetNearbyObjEq` is what Joe needs. There are several challenges with showing such a recommendation. First, the recommended snippet is not a table, it is a user-defined function. Second, the desired tables, views, or UDFs are not necessarily popular by themselves. They are just frequently used in the context of the query that the user wrote so far. Finally, all recommendations must be done at interactive speed for the user to remain focused.

Additionally, upon seeing a recommendation, a user can be confused as to how to use the recommended snippet. To address this challenge, SnipSuggest can show, upon request, either documentation related to the suggested snippet, or real queries that use it.

After this first step, Joe's query thus looks as follows:

```
SELECT *
FROM PhotoPrimary P,fGetNearbyObjEq(145.622,0.0346249,2) n
WHERE
```

Joe would now like to restrict the objects to include only those with a certain redness value. Encouraged by his early success with SnipSuggest, instead of browsing through documentation again, he directly asks SnipSuggest for recommendations for his `WHERE` clause. First is the missing foreign-key join `p.objId = n.objId`. Once added, SnipSuggest's next recommendations become: `p.dec<#`, `p.ra>#`, `p.dec>#`, `p.ra<#`, and `p.r≤#`. These predicates are the most popular predicates appearing in similar past queries. After a quick glance at `p.r`'s documentation, Joe picks the last option, and adds the following predicate to his query: `p.r < 18 AND p.r > 15`. This example demonstrates the need for the query log. With the exception of foreign-key joins, it is not possible to determine useful predicates for the `WHERE` clause using the database schema alone. Past queries enable SnipSuggest to select the relevant predicates among the large space of all possible predicates.

Now, his query looks as follows:

```
SELECT *
FROM PhotoPrimary P,fGetNearbyObjEq(145.622,0.0346249,2) n
WHERE p.objID = n.objID AND p.r < 18 AND p.r > 15
```

In a similar fashion, SnipSuggest can help Joe to add a second predicate (i.e., keep only objects that are stars: `p.type = 6`), and to write the `SELECT` and `GROUP BY` clauses.

In summary, the challenges for SnipSuggest are to

1. recommend relevant features without knowledge of the user's intended query,

2. leverage the current query context to improve the recommendation quality, and

3. produce recommendations efficiently.

3.3 System Architecture

SnipSuggest is a middleware-layer on top of a standard relational DBMS as shown in Figure 3.1. While users submit queries against the database, SnipSuggest's Query Logger component logs these queries in a Query Repository. Upon request, SnipSuggest's Snippet Recommender uses this query repository to produce SQL-autocomplete recommendations. Finally, SnipSuggest's Query Eliminator periodically prunes the query log to improve recommendation performance by shrinking the Query Repository. We now present these three components and SnipSuggest's algorithms.

3.3.1 Query Logger and Repository

When the Query Logger logs queries, it extracts various *features* from these queries. Informally, a feature is a specific fragment of SQL such as a table name in the `FROM` clause (or

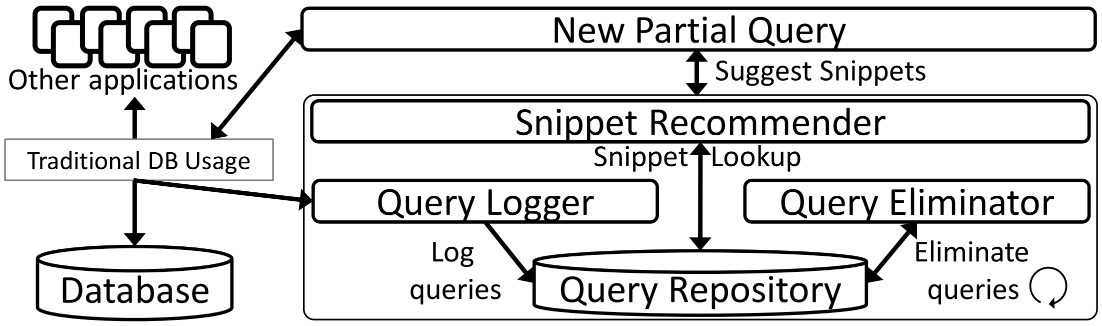


Figure 3.1: SnipSuggest system architecture.

view name or table-valued function name), or a predicate in the `WHERE` clause. The Query Logger is implemented on top of the existing infrastructure for query logging offered by most DBMSs.

The Query Repository component stores the details of all the queries logged by the Query Logger, along with the features which appear in each query. It comprises three relations: `Queries`, `Features`, and `QueryFeatures`, with the following schemas:

1. `Queries(id, timestamp, user, database name, query text, running time, output size)`

2. `Features (id, feature description, clause)`

3. `QueryFeatures(query, feature)`

The first relation, named `Queries`, stores the details of each logged query. The second relation, `Features`, consists of all the features that have been extracted from these queries. Note that feature descriptions are parameterized if there is some constant involved (e.g., the predicate `PhotoPrimary.objID = 55` is translated into the parameterized predicate `PhotoPrimary.objID = #`). Finally, the third table, `QueryFeatures`, maintains the information about which feature appears in which query. The `query` and `feature` columns are foreign keys into the `Queries` and `Features` tables, respectively.

Features Supported

The current implementation of SnipSuggest supports the following classes of features:

1. F_T^{from} for every table, view and table-valued function T in the database, representing whether T appears in the `FROM` clause of the query.

2. F_C^{select}, F_C^{where}, and $F_C^{groupby}$, for every column C in the database, representing whether this column appears in the `SELECT`, `WHERE`, or `GROUP BY` clause of the query, respectively.

3. $F_{aggr(C_1,\ldots C_n)}^{select}$, for every aggregate function and list of columns, representing whether this aggregate and list of columns appear in the `SELECT` clause.

4. $F_{C_1\ op\ C_2}^{where}$ for every pair of columns C_1, C_2, and every operator which appears in the database, representing whether this predicate appears in the `WHERE` clause of the query.

5. $F_{C\ op}^{where}$ for every column C in the database, and for every operator, representing whether there is a predicate of the form C *op constant* in the `WHERE` clause of the query.

6. $F_{ALL}^{subquery}$, $F_{ANY}^{subquery}$, $F_{SOME}^{subquery}$, $F_{IN}^{subquery}$, and $F_{EXISTS}^{subquery}$ representing whether there is a subquery in the `WHERE` clause, of the form `ALL(subquery)`, `ANY(subquery)`, `SOME(subquery)`, `IN(subquery)`, `EXISTS(subquery)`, respectively.

3.4 Snippet Recommendation

While a user composes a query, she can, at any time, select a clause, and ask SnipSuggest for recommendations in this clause. At this point, SnipSuggest's goal is to recommend k features that are most likely to appear in that clause in *the user's intended query.*

To produce its recommendations, SnipSuggest views the space of queries as a directed acyclic graph (DAG) [1] such as that shown in Figure 3.2 (which we return to later). For this, it models each query as a set of features and every possible set of features becomes a vertex in the DAG. When a user asks for a recommendation, SnipSuggest, similarly, transforms the user's partially written query into a set of features, which maps onto a node in the DAG. Each edge in the DAG represents the addition of a feature (i.e., it links together sets of

[1]Note that the DAG is purely a conceptual model underlying SnipSuggest. The user never interacts with it directly.

features that differ by only one element). The recommendation problem translates to that of ranking the outgoing edges for the vertex that corresponds to the user's partially written query, since this corresponds to ranking the addition of different features.

The query that the user intends to write is somewhere below the current vertex in the DAG, but SnipSuggest does not know which query it is. It approximates the intended query with the set of all queries in the Query Repository that are descendants of the current vertex in the DAG. We refer to such queries as the *potential goals* for the partial query. For now, we assume that the set is not empty (and discuss the alternative at the end of Section 3.4.4). Given this set of *potential goals*, there are several ways to rank the features that could possibly be added to the user query. We investigate two of them in this project. The first approach is simply to recommend the most popular features among all those queries. The problem with this approach is that it can easily lead to k recommendations all leading to a single, extremely popular query. An alternate approach is thus to recommend k features that cover a maximal number of queries in the *potential goals* set.

We now describe the problem and our approach more formally.

3.4.1 Definitions

We begin with the definition of features.

Definition 1 *A* **feature** f *is a function that takes a query as input, and returns true or false depending on whether a certain property holds on that query.*

Some examples are $f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}$, representing if the `PhotoPrimary` table appears in the query's `FROM` clause, $f^{\texttt{WHERE}}_{\texttt{PhotoPrimary.objID=Neighbors.objID}}$, representing whether the predicate `PhotoPrimary.objID = Neighbors.objID` appears in the `WHERE` clause, or $f_{\texttt{distinct}}$ representing whether the `distinct` keyword appears anywhere in the query. A feature can have a clause associated with it, denoted $clause(f)$. For example, $clause(f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}) = \texttt{FROM}$. Through this project, we use the notation f^c_s to denote the feature that string s appears in clause c.

Definition 2 *The* **feature set of a query** q*, is defined as:*

$$features(q) = \{f | f(q) = true\}$$

When SnipSuggest 'recommends a snippet', it is recommending that the user modify the query so that the snippet evaluates to true for the query. For example, when it recommends $f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}$, it is recommending that the user add `PhotoPrimary` to the `FROM` clause.

Definition 3 *The* **dependencies of a feature** f*, dependencies*(f)*, is the set of features that must be in the query so that no syntactic error is raised when one adds* f*.*

e.g., $dependencies(f^{\texttt{WHERE}}_{\texttt{PhotoPrimary.objID=Neighbors.objID}})=\{f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}, f^{\texttt{FROM}}_{\texttt{Neighbors}}\}$. SnipSuggest only suggests a feature f for a partial query q if $dependencies(f) \subseteq features(q)$. In the workload DAG, feature sets have parent-child relationships defined as follows:

Definition 4 *A feature set* F_2 *is a* **successor** *of a feature set* F_1*, if* $\exists\ f$ *where* $F_2{=}F_1{\cup}\{f\}$ *and dependencies*$(f){\subseteq}F_1$*.*

A successor of a feature set F_1 is thus a feature set F_2 that can be reached by adding a single, valid feature.

Additionally, recommendations are based on feature popularity that is captured by either marginal or conditional probabilities.

Definition 5 *Within a workload* W*, the* **marginal probability of a set of features** F *is defined as*

$$P(F) = \frac{|\{q \in W | F \subseteq features(q)\}|}{|W|}$$

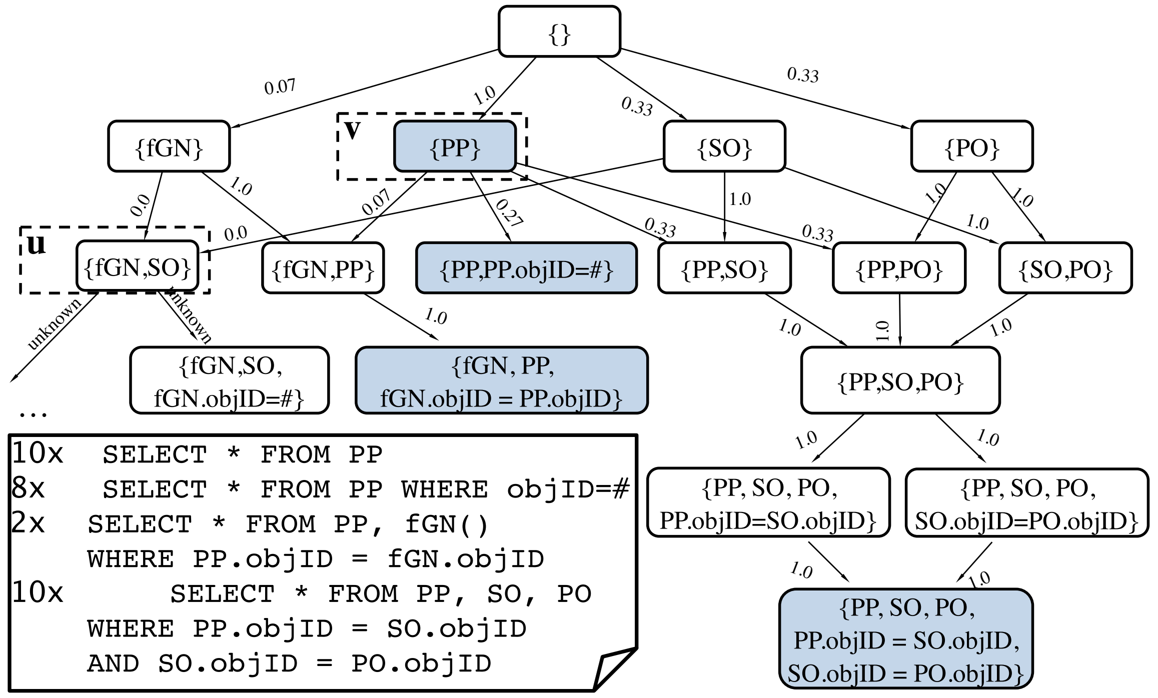
i.e. the fraction of queries which are supersets of F*. As shorthand, we use* $P(Q)$ *for* $P(features(Q))$*, and* $P(f)$ *for* $P(\{f\})$*.*

Definition 6 *The* **conditional probability of a feature** f **given a feature set** F *is defined as*

$$P(f|F) = \frac{P(\{f\} \cup F)}{P(F)}$$

We are now ready to define the workload DAG. Let F be the set of all features (including those that do not appear in workload).

Definition 7 *The* **workload DAG** $T = (V, E, w, \chi)$ **for a query workload** W *is constructed as follows:*



```
10x  SELECT * FROM PP
8x   SELECT * FROM PP WHERE objID=#
2x  SELECT * FROM PP, fGN()
    WHERE PP.objID = fGN.objID
10x     SELECT * FROM PP, SO, PO
    WHERE PP.objID = SO.objID
    AND SO.objID = PO.objID
```

Figure 3.2: Example of a workload DAG.

1. *Add to V, a vertex for every syntactically-valid subset of F. We refer to each vertex by the subset that it represents.*
2. *Add an edge (F_1, F_2) to E, if F_2 is a successor of F_1. Denote the additional feature of F_2 by* $\textbf{addlFeature}((\mathbf{F_1}, \mathbf{F_2})) = f$, *where* $F_2 = F_1 \cup \{f\}$.
3. $w : E \to [0, 1]$ *is the weight of each edge. The weights are set as:* $w((X, Y)) = P(addlFeature((X, Y))|X)$. *If* $P(X) = 0$, *then set to* `unknown`.
4. $\chi : V \to \{blue, white\}$ *is the color of each vertex. The colors are set as:* $\chi(q) = blue$ *if* $q \in W$, *otherwise white.*

Figure 3.2 shows an example workload DAG for 30 queries. The queries correspond to the blue nodes, and are summarized at the bottom. Ten are of the form `SELECT * FROM PhotoPrimary`, eight are `SELECT * FROM PhotoPrimary WHERE objID = #`, etc. For simplicity, we exclude, from the figure, features in the `SELECT` clause, and nodes that are not reachable from the root along edges of weight > 0, with the exception of node u. We use the acronyms fGN, PP, SO, and PO to represent $f^{\texttt{FROM}}_{\texttt{fGetNearbyObjEq}}$, $f^{\texttt{FROM}}_{\texttt{PhotoPrimary}}$, $f^{\texttt{FROM}}_{\texttt{SpecObjAll}}$, and $f^{\texttt{FROM}}_{\texttt{PhotoObjAll}}$, respectively. The edge ($\{PP\}$, $\{PP, SO\}$) indicates that if a query contains PhotoPrimary, there is 33% chance that it also contains SpecObjAll.

Every syntactically-correct partial query appears in the workload DAG since there is a vertex for every valid subset of F. Consider a partial query, and its corresponding vertex q. Given q, SnipSuggest's goal is to lead the user towards their intended query $q*$ (also a vertex in the DAG), one snippet at a time. We assume that there is a path from q to $q*$, i.e., that the user can reach $q*$ by adding snippets to their query. Since SnipSuggest suggests one snippet at a time, the recommendation problem becomes that of ranking the outgoing edges of q. Note that recommending an edge e corresponds to recommending $addlFeature(e)$.

For example, suppose Anna has written: SELECT * FROM PhotoPrimary. This puts her at vertex v in Figure 3.2. Then, she requests snippets to add to the FROM clause. At v, we see that she can add fGetNearbyObjEq(), SpecObjAll, or PhotoObjAll. Remember, Figure 3.2 is not showing the whole DAG. In fact, the full workload DAG contains an outgoing edge from v for each of the 342 tables, views, and table-valued functions in the SDSS schema. The job of SnipSuggest is to recommend the edges that are most likely to lead Anna towards her intended query.

3.4.2 Naïve Algorithms

We present three techniques that we compare against SnipSuggest in Section 3.6. The most naïve, the **Random** recommender, ranks the outgoing edges randomly. The second approach, **Foreign-key-based**, used only for suggesting snippets in the WHERE clause, exploits the schema information to rank the features. It suggests predicates for foreign-key joins before other predicates. The third approach, the **Popularity-based** technique actually leverages the past workload. It considers $f = addlFeature(e)$ for each outgoing edge from q, and ranks them by $P(f)$, the marginal probability of f. Even this simple technique, outperforms the above two algorithms by up to 449%.

The problem with these approaches is that they do not exploit the rich information available in the workload DAG; the weighted edges can tell us which features are likely to appear in the intended query, given the current partial query. SnipSuggest's algorithms aim to better recommend snippets by leveraging such information.

3.4.3 Context-Aware Algorithms

In this section, we introduce the two algorithms that SnipSuggest uses to recommend suggestions based on the current context (i.e., the current partial query). We describe the algorithms in more detail in the subsequent sections (Section 3.4.4 and Section 3.4.5).

First, we need one more notion, and a precise definition of the problem that each algorithm aims to solve.

Definition 8 *Given a workload DAG and a vertex q, define:*
$potential\_goals(q) = \{v | v \text{ is blue and reachable from } q\}$.

The *potential_goals* of q is the set of queries that could potentially be the user's intended query, if it appears in the workload. Sometimes, $potential\_goals(q)$ is the empty set.

We consider two variations of the Snippet Suggestion Problem. Given a workload DAG G, and a partial query q, recommend a set of k outgoing edges, $e_1, \ldots, e_k$, from q that:

1. **Max-Accuracy Problem:** maximizes

$$\sum_{i=1}^{k} P(addlFeature(e_i)|q)$$

2. **Max-Query-Coverage Problem:** maximizes

$$P(addlFeature(e_1) \vee \ldots \vee addlFeature(e_k)|q)$$

Max-Accuracy aims to maximize the number of features in the top-k that are helpful (i.e., appear in the intended query), whereas Max-Query-Coverage aims to maximize the probability that at least one feature in the top-k is helpful.

Consider the earlier example. Suppose SnipSuggest recommends the top-2 snippets to add to the `FROM` clause. If the goal is Max-Accuracy, then it suggests `SO` and `PO`. This corresponds to the two outgoing edges from q, with the highest conditional probabilities. If the aim is Max-Query-Coverage, then it suggests `SO` and `fGN`. The reasoning is as follows: if Anna's intention is the rightmost blue query, then suggesting `SO` covers this case. If her intention is not that query, then rather than `PO`, it is better to suggest `fGN` because it increases the number of *potential_goals* covered.

It is infeasible to build the workload DAG as it can have up to 2^n vertices, where $n = |F|$. Thus, SnipSuggest implements two algorithms, which simulate traversing parts of the DAG, without ever constructing it: *SSAccuracy* and *SSCoverage*.

3.4.4 SSAccuracy

Given a partial query q and a query workload W, the goal of the *SSAccuracy* algorithm is to suggest the k features with the highest conditional probabilities given q. If q's features have previously appeared together in past queries, *SSAccuracy* is able to efficiently identify the features with the highest conditional probabilities, with a single SQL query over the *QueryFeatures* table, as shown in Figure 3.3. By setting m to $|features(q)|$, the first half of the query finds $potential\_goals(q)$, i.e. the queries which have all the features of q. It then orders all the features which appear in these `SimilarQueries`, by their frequencies within this set of queries. Note that each `qf.feature` f corresponds to one outgoing edge from q (i.e. the edge e where $addlFeature(e) = f$). Additionally, if we divided `count(s.query)` by $|potential\_goals(q)|$, we would find $P(f|q)$. Thus, this query returns a list of edges, ordered by weight (i.e., the conditional probability of the feature given q).

What if the partial query q does *not* appear in the workload? Every partial query q appears in the DAG, but it can happen that all incoming edges have weight 0, and all outgoing edges are `unknown`. This happens when $potential\_goals(q) = \emptyset$. An example of this, in the context of the workload DAG in Figure 3.2, is if Bob has written $q =$ `SELECT * FROM SpecObjAll, fGetNearbyObjEq(143.6,0.021,3)`, and requests suggestions in the `FROM` clause (which is represented by vertex u in the figure).

In this case, SnipSuggest traverses up the DAG from q until it reaches the vertices whose marginal probability is not zero (i.e., there exists an incoming edge with weight > 0). This corresponds to finding the largest subsets of $features(q)$ that appear in the workload. In the

```
WITH SimilarQueries (query) AS --finds potential_goals
(SELECT query
 FROM  QueryFeature
 WHERE  feature IN features(q)
  [AND NOT EXISTS ( --used only by SSCoverage
   select * from QueryFeature q
   where q.query=query and q.feature in( previous))]
 GROUP BY query
 HAVING count(feature) =  m)
SELECT qf.feature --popular features among SimilarQueries
FROM   QueryFeature qf, SimilarQueries s
WHERE qf.query = s.query AND qf.feature NOT IN features(q)
GROUP BY qf.feature
ORDER BY count(s.query) DESC
```

Figure 3.3: Finds the most popular features among queries that share m features with partial query q. `NOT EXISTS` clause is included for SSCoverage, but omitted for SSAccuracy.

above example, SnipSuggest traverses up to the vertices {`SO`} and {`fGN`}. Then, it suggests the most popular features among the queries under these vertices. This can be achieved by executing the SQL query shown in Figure 3.3. First, SnipSuggest sets m to $|features(q)|$, thus looking at $potential\_goals(q)$. If fewer than k features are returned, then it sets m to $|features(q)| - 1$, thus considering queries that share $|features(q)| - 1$ features with q. It repeatedly decrements m until k features are returned. Note that SnipSuggest executes the query in Figure 3.3 at most $|features(q)|$ times. In other words, SnipSuggest will not iterate through every subset of $features(q)$. Instead, it considers all subsets of size m all at once, and it does this for $m = n, n-1, n-2, \ldots, 0$, where $n = |features(q)|$.

This process can cause some ambiguity of how to rank features. For example, if it is the case that $P(f_1|\{SO, fGN\}) = 0.8$ and $P(f_2|\{SO\}) = 0.9$, it is not clear whether

Algorithm 1 SnipSuggest's Suggestion Algorithm.

Input: query q, number of suggestions k, clause c, technique t

Output: a ranked list of snippet features

```
1: $i \leftarrow |features(q)|$
2: $suggestions \leftarrow []$
3: while $|suggestions| < k$ : do
4:   if $t = SSAcc$ then
5:     $S \leftarrow$ execute Figure 3.3 query ($m \leftarrow i$, exclude NOT EXISTS clause)
6:   else if $t = SSCov$ then
7:     $S \leftarrow$ execute Figure 3.3 query ($m \leftarrow i$, $previous \leftarrow suggestions$)
8:   end if
9:   for all $s \in S$ do
10:    if $s \notin suggestions$ and $clause(s) = c$ then
11:      $suggestions \leftarrow suggestions, s$
12:    end if
13:  end for
14:  $i \leftarrow i - 1$
15: end while
16: return $suggestions$
```

f_1 or f_2 should be ranked first. Heuristically, SnipSuggest picks f_1, because it always ranks recommendations based on more similar queries first. Algorithm 1 outlines the full SSAccuracy algorithm (if we pass it $t = SSAcc$). In Section 3.6, we show that *SSAccuracy* achieves high average precision, and we now describe two simple optimizations.

Simple Optimizations

SnipSuggest materializes the following two relations to improve the SSAccuracy algorithm's recommendation time.

1. `MarginalProbs(featureID, probability)`

2. `CondProbs(feature1, feature2, probability)`

The first table stores the marginal probability for each feature across the whole workload. The second contains the conditional probability of `feature1` given `feature2`, for every pair of features that have ever appeared together. We now discuss how these tables are used.

The first is $MarginalProbs$, which contains $P(f)$ for every feature f. When the user's partial query q is the empty query (i.e., $features(q) = \emptyset$), or if q consists of only features that have never before appeared in the workload, SnipSuggest can execute an `order by` query over $MarginalProbs$, instead of the more complex SQL in Figure 3.3. (When q contains only unseen features, SnipSuggest traverses up to the root vertex since it is the largest subset of q that appears in the workload. So, SnipSuggest makes its suggestions for $\emptyset$, and thus exploits $MarginalProbs$.)

The second is $CondProbs$, which contains the conditional probability $P(f_1|f_2)$ for every pair of features f_1, f_2. It is indexed on the f_2 column. It is leveraged when the user's partial query q contains just one feature f, or it contains multiple features, but only one feature f has appeared in the workload. In these cases, SnipSuggest can execute a simple query over $CondProbs$ with filter $f_2 = f$, and order by the conditional probability, instead of executing the slower SQL in Figure 3.3.

3.4.5 SSCoverage

The Max-Query-Coverage problem is to suggest the features $f_1, \ldots, f_k$ that maximize the probability that at least one suggestion is helpful. The goal is to diversify the suggestions, to avoid making suggestions that all lead toward the same query. It turns out that the problem is NP-hard. Instead of an exact solution, we propose an approximation algorithm, SSCoverage, which is a greedy, approximation algorithm for Max-Query-Coverage. We prove at the end of this section that Max-Query-Coverage is NP-hard and that SSCoverage is the best possible approximation for it. We show this by proving that Max-Query-Coverage is equivalent to the well-known Maximum Coverage problem [54]. Our equivalence proof is sufficient because it is known that the Maximum Coverage problem is NP-hard, and that the best possible approximation is the greedy algorithm [54], achieving an approximation factor of $1 - \frac{1}{e}$.

The *SSCoverage* algorithm proceeds as follows. To compute the first recommendation f_1, it executes the SQL query in Figure 3.3, with the `NOT EXISTS` clause and `previous` $\leftarrow \emptyset$. This is equivalent to *SSAccuracy*'s first recommendation because the Max-Accuracy and Max-Query-Coverage formulas are equivalent when $k = 1$. For its second suggestion, *SSCoverage* executes the Figure 3.3 query again, but with `previous` $\leftarrow \{f_1\}$. This effectively removes all the queries covered by f_1 (i.e., $potential\_goals(q \cup \{f_1\})$), and finds the feature with the highest coverage (i.e., conditional probability) in the remaining set. In terms of the workload DAG, this step discards the whole subgraph rooted at f_1, and then finds the best feature among the remaining DAG. It repeats this process k times in order to collect k features. Algorithm 1 describes SSCoverage in detail (if we pass it $t = SSCov$).

Max-Query-Coverage Related Proofs

In this section, we show that the Max-Query-Coverage problem is NP-hard, and that the SSCoverage algorithm is the best possible approximation algorithm for it (up to lower order terms).

Remember, the Max-Query-Coverage problem, as presented in Section 3.4.3, is defined as follows. Given a workload DAG G and a partial query q, recommend a set of k outgoing edges, $e_1, \ldots, e_k$, from q that maximizes

$$P(addlFeature(e_1) \vee \ldots \vee addlFeature(e_k) | q)$$

For shorthand, denote by f_i the feature $addlFeature(e_i)$. To make our next step easier, we want to show that the features $f_1, \ldots, f_k$ that maximize the formula above are the ones that maximize the number of queries covered, under the q vertex.

Consider $P(f_1 \vee \ldots \vee f_k | q)$. If we select some random query whose feature set is a superset of $features(q)$ (i.e. any query in $potential\_goals(q)$), then this is the probability that it also has feature f_1, f_2, …, or f_k. So, $P(f_1 \vee \ldots \vee f_k | q)$ can be written as:

$$\sum_{u \in potential\_goals(q)} Pr(f_1 \vee \ldots \vee f_k | q \wedge u) \cdot Pr(u|q)$$

$Pr(f_1 \vee \ldots \vee f_k | q \wedge u)$ is 1 if u contains f_1, f_2, …, or f_k, and 0 otherwise (because q does not contain any of $f_1, \ldots, f_k$ either).

Hence this is equal to:

$$\sum_{u \in potential\_goals(q):(f_1 \in u \vee \ldots \vee f_k \in u)} Pr(u|q)$$

Therefore, we can now rewrite the Max-Query-Coverage problem to maximize:

$$\sum_{v \in \ U} P(v|q),\ where\ U = \bigcup_{i=1}^{k} potential\_goals(q \cup \{f_i\})$$

Next, we define the Maximum Coverage Problem, which is known to be NP-hard [54].

Definition 9 *Given a set of elements U, a number k and a set of sets $S = S_1, \ldots S_n$, where each $S_i \subseteq U$, the* **maximum coverage problem** *is to find a subset of sets $S' \subseteq S$ such that $|S'| \leq k$ and the following is maximized:*

$$|\bigcup_{S_i \in S'} S_i|$$

i.e., the number of elements covered is maximized.

In his paper [54], Feige proves the following theorem.

Theorem 1 *The maximum coverage problem is NP-hard to approximate within a factor of $1 - \frac{1}{e} + \epsilon$, for any $\epsilon > 0$.*

Moreover, the greedy algorithm achieves an approximation factor of $1 - \frac{1}{e}$. The analysis of the greedy algorithm was well-known, and is reproved in Feige's paper [54].

We prove our results by showing Max-Query-Coverage's equivalence to the Maximum Coverage Problem.

Theorem 2 *The Max-Query-Coverage is equivalent to the Maximum Coverage problem.*

In particular, there are polynomial time reductions between the two problems, which preserve the values of all solutions. With this theorem, and the known results described above, we can conclude that Max-Query-Coverage is NP-hard to approximate within any factor substantively better than $1-\frac{1}{e}$, and that SnipSuggest's greedy algorithm, SSCoverage, achieves this bound.

Proof 1 (of Theorem 2) *We show two mappings. First, we show that an instance of the Maximum Coverage Problem can be mapped to an instance of the Max-Query-Coverage problem, and that there is a bijection between the set of solutions to each. Second, we show the reverse.*

Consider an instance I_1 of the Maximum Coverage Problem, where U is the set of elements and $S = S_1, \ldots, S_m$ is the collection of sets. We translate I_1 into an instance I_2 of the Max-Query-Coverage problem as follows. U is the set of queries, S is the set of features. Denote by f_T the feature that corresponds to the set T. T represents the set of queries which contain the feature f_T. For a given element/query $q \in U$, $features(q) = \{f_T : q \in T\}$. The input partial query is the empty query, and so the problem is to suggest the top-k features given the empty query. Next, we show that there is a bijection between the solutions for I_1 and I_2.

Given any solution $S' = \{S'_1, \ldots, S'_k\}$ to I_1 (not necessarily an optimal solution), the equivalent solution in I_2 is to suggest the features $F' = \{f_{S'_1}, \ldots, f_{S'_k}\}$. Of course, conversely, given a solution $F' = \{f'_1, \ldots, f'_k\}$ to I_2, the equivalent in I_1 is $S' = \{S'_i : f'_{S_i} \in F'\}$. Clearly, the number of elements of U covered by S' is the same as the number of queries covered by F' (which is the mass covered by F' multiplied by $|W|$).

Now, consider an instance I_1 of the Max-Query-Coverage problem, where the query workload is W, the set of all features is F, and the partial query is q. We translate this into the Maximum Coverage Problem as follows. Scan through W to find $potential\_goals(q)$ (i.e., all the blue vertices under q). For each potential goal q', if it is a leaf node, add $P(q') \times |W|$ elements to U. If it is not a leaf, add $(P(q') - \sum_{f \in F} P(q' \cup \{f\})) \times |W|$ elements. This represents the number of queries in the workload that have exactly this set of features. Note that the number of elements added to U is at most the number of queries in W. Add to S, one set per feature $f \in F$, consisting of all queries that contain f. Let's denote this by $queries(f)$.

Given any solution $F' = \{f_1, \ldots, f_k\}$ to I_1 (not necessarily optimal), the equivalent solution in I_2 is to select the sets $S' = \{queries(f_1), \ldots, queries(f_k)\}$. Conversely, a solution $S' = \{S'_1, \ldots, S'_k\}$ to I_2 can be translated to the following solution for I_1: $F' = \{f'_i : queries(f'_i) = S'_i\}$. If we consider the number of queries covered by F', this is equal to the *number of elements of U covered by S'.*

We have shown that we can translate an instance of the Maximum Coverage Problem into an instance of the Max-Query-Coverage problem, and the reverse, in time polynomial in $|W|$. We've also shown that given an instance of one problem, and its corresponding instance in the other, there is a bijection between the solutions for the two instances.

3.5 Query Elimination

All queries, correct or incorrect, are logged by the Query Logger. This is problematic for SnipSuggest because a large workload can deteriorate its response time. The Query Eliminator addresses this problem; it periodically analyzes the most recent queries and drops some of them. The goal is to reduce the workload size, and the recommendation time, while maintaining recommendation quality.

For the Query Eliminator, we introduce the notion of a query session.

Definition 10 *A* query session *(or* session *for short) is a sequence of queries $q_1, \ldots, q_n$ written by the same user as part of a single task. We refer to the last query q_n as the* target query*.*

The Query Eliminator eliminates all queries, except those that appear at the end of a session. Intuitively, queries that appear near the end of the session are of higher quality, since the user has been working on them for longer. Since many users write a handful of queries before reaching their intended one, this technique eliminates a large fraction of the workload. We show in Section 3.6.3 that the Query Eliminator reduces the response time, while maintaining, and often improving, the average precision.

SnipSuggest extracts query sessions in two phases (as shown in Figure 3.4). First, SnipSuggest segments the incoming query log. It does so by monitoring changes between consecutive queries in order to detect starts of new revision cycles. A *revision cycle* is the iterative process of refining and resubmitting queries until a desired task is complete. When it detects a new cycle, SnipSuggest labels, as query segment, the set of all queries since the beginning of the previous cycle. We call this phase *segmentation.* We describe segmentation in more detail in Section 3.5.1. Second, SnipSuggest stitches multiple segments together, if

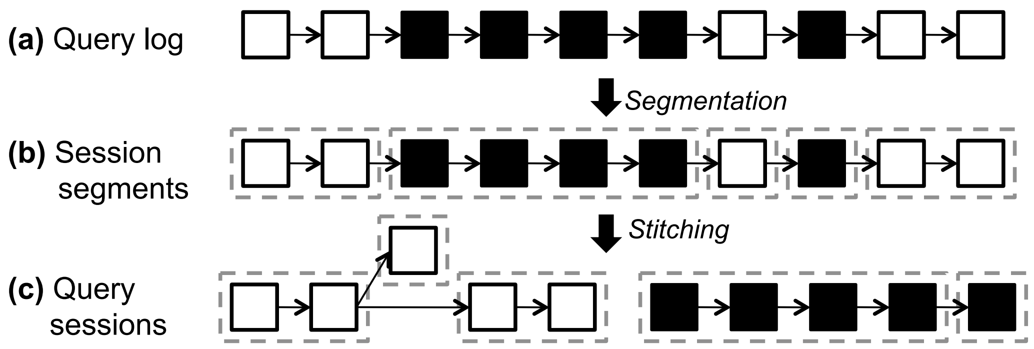


Figure 3.4: Extracting query sessions from the query log.

they are part of a single, larger revision cycle. For example, when a user is stuck on a difficult task A, they often move to a different task B and later return to A. In this scenario, A will produce multiple query segments, because the queries for task B will separate the later queries of A from the earlier ones. Via *stitching*, the segments are concatenated together to create a large session for A. We present the details of the stitching phase in Section 3.5.2.

Both phases require an expert to provide some training data, in the form of a query log with labeled sessions. SnipSuggest leverages machine learning techniques to learn the appropriate thresholds for the segmentation and stitching.

3.5.1 Segmentation

The goal of the **segmentation phase** is to take a pair of *consecutive* queries P, Q, and decide whether the two queries belong to the same query session or not. The algorithm proceeds in three steps. First, it constructs the Abstract Syntax Tree (AST) for each query and transforms it into canonical form, which includes, for example, removing any constants, and alphabetically ordering the list of tables in the `FROM` clause (this process is also used by the Query Logger). Second, it extracts a set of segmentation features from P and Q as well as extra information such as timestamps of queries and the query output of the preceding query P. Unlike SnipSuggest features, the segmentation features capture the difference between two queries. Some examples include the time interval between the queries, the cosine similarities between their different clauses, and the relationship between ASTs. In the third step, using these segmentation features, our technique uses a perceptron-based classification algorithm to decide whether the queries belong in the same segment. For this final step, as training data, SnipSuggest requires some labeled data in the form of queries labeled with the identifier of the task that the queries are intended for.

The most significant segmentation feature is *AST inclusion type*. This feature represents whether the relationship between the two queries' ASTs is the **Same**, **Add**, **Delete**, **Merge**, **Extract** or **None**. This feature captures the following intuition. Within a query segment, the user incrementally adds or removes terms from the query after seeing the query result of the previous query. Such incremental edits are captured by the values **Same**, **Add**, or **Delete**. Occasionally, the user may introduce a subquery written in the past or copied from some sample query to compose a more complex query. The user may also pull-out a subquery to debug or analyze unexpected results. In both these cases, the user may start to work towards a different purpose when the change involves subqueries; this signals a new query segment. **Merge** and **Extract** captures such changes involving subqueries. Table 3.1 summarizes these five AST inclusion types.

Although the *AST inclusion type* may be a strong indicator of continuing or breaking a query segment, it does not capture the amount of change. Thus, the other segmentation features that we described above, which capture the degree of change, are necessary for better accuracy.

Label	Description	Expect a new segment?
Same	Q is canonically the same as Q	No
Add	Q is based on P and has more terms	No
Delete	Q is based on P and has fewer terms	No
Merge	P is a subquery of Q	Yes
Extract	Q is a subquery of P	Yes
None	All other types of changes	Unknown

Table 3.1: Summary of AST inclusion types. Each feature value has an expected decision on segmentation discussed in the text. P and Q denote preceding query and following query, respectively.

3.5.2 Stitching

Query segments are useful because of the coherency among the queries in the same segment. However, in order to extract more complete query sessions, SnipSuggest often needs to stitch together multiple segments.

The **stitching phase** tests whether two segments create a single revision cycle, smooth transitions via small changes, when they are concatenated in time order. To perform stitching, SnipSuggest iterates over each segment s in the input and all its time-wise successor segments. It runs the core of the segmentation algorithm between the last query of s and the first query of a time-wise successor segment t, with a modification. Since SnipSuggest is now considering segments that are separated by multiple segments, the time interval between the queries becomes less meaningful. Thus, the algorithm treats the time interval as a missing attribute. Then, if the segmentation algorithm outputs that the last query of s and the first query of t belong in the same segment and this does not contradict to value of *AST inclusion type*, SnipSuggest concatenates the two segments.

3.6 Evaluation

We evaluate SnipSuggest over two datasets. The first consists of queries from the Sloan Digital Sky Survey. The SDSS database logs all queries submitted through public interfaces including web pages, query forms, and a web services API. Thus, query authors vary from bots, to the general public, and real scientists. We downloaded 106 million queries from 2002 to 2009. Removing queries from bots, ill-formed queries, and queries with procedural SQL or proprietary features, left us with approximately 93 million queries. For our evaluation, we use a random sample of 10,000 queries, in which there are 49 features for tables, views and table-valued functions, 1835 columns and aggregates, and 395 predicates.

The second dataset consists of SQL queries written by students in an undergraduate database class, which were automatically logged as they worked on nine different problems for one assignment. All queries are over a local copy of the Internet Movie Database (IMDB), consisting of five tables and 21 columns. To evaluate, we use a sample consisting of ten students' queries, which results in a total of 1679 queries. For each student, we manually label each query with the assignment problem number that it was written for, which gives us ground truth information about query sessions and thus serves as training data for the Query Eliminator.

We aim to answer four questions: Is SnipSuggest able to effectively recommend relevant snippets? Does its recommendation quality improve as the user adds more information to a query? Can it make suggestions at interactive speeds? Is the Query Eliminator effective at reducing response times, while maintaining recommendation quality? We evaluate the first three questions on both datasets. The fourth is answered on only the IMDB dataset because we do not have the ground truth for the SDSS query sessions.

3.6.1 Evaluation Technique

Neither query log includes "partial queries"; they only include full SQL queries that were submitted for execution. Therefore, in order to evaluate SnipSuggest on a given query, we remove some portion of the query, and the task is to recommend snippets to add to this new partial query. We denote by $fullQuery(q)$, the full SQL query from which the partial query q was generated. For this setting, we define correctness for a partial query q, and feature f.

Definition 11 *For a given partial query q, a suggested snippet feature f is* correct *if and only if $f \notin features(q)$ and $f \in features(fullQuery(q))$.*

To measure the recommendation quality of SnipSuggest, we use a measure called average precision [25]. It is a widely-used measure in Information Retrieval for evaluating ranking techniques. SnipSuggest returns a ranked list of snippets, L_q, for query q.

Definition 12 *The average precision at k for the suggestions L_q is*

$$AP_{@k}(q, L_q) = \frac{\sum_{i=1}^{k}(P(q, L_q, i) \cdot rel(q, L_q[i]))}{|features(fullQuery(q)) - features(q)|}$$

where $P(q, L_q, k)$ is the precision of the top-k recommendations in L_q and $rel(q, L_q[i]) = 1\ if L_q[i]$ is correct, and 0 otherwise. Precision is defined as $P(q, L_q, k) = \frac{\sum_{i=1}^{k} rel(q, L_q[i])}{k}$

This measure looks at the precision after each correct snippet is included, and then takes their average. If a correct snippet is not included in the top-k, it contributes a precision of zero. The benefit of using this approach, instead of recall or precision, is that it rewards the techniques that put the correct snippets near the top of the list.

Consider the following toy example of the Average Precision measure in action. The aim is to give the reader a better understanding of Average Precision.

Suppose that Carol has an empty query, and has requested the top-5 suggestions for the `FROM` clause. Her intended query includes two features in the `FROM` clause: `PhotoPrimary` and `fGetNearbyObjEq()`. Suppose that SnipSuggest has suggested the following snippets, in this order: `PhotoPrimary`, `SpecObjAll`, `fGetNearbyObjEq()`, `PhotoObjAll`, `Columns`.

Consider the following table, which will help us calculate the Average Precision of these suggestions.

Rank	Suggestion	Correct?	Precision
1	`PhotoPrimary`	true	1.0
2	`SpecObjAll`	false	0.5
3	`fGetNearbyObjEq()`	true	0.67
4	`PhotoObjAll`	false	0.5
5	`Columns`	false	0.33

The *Precision* column at rank i indicates the precision of the first i suggestions. With this table, we collect the ranks where the suggestion is correct. In this example, these are ranks 1 and 3. Then, we take the sum of the precisions at these ranks and divide by two (i.e., the number of ground truth features). For this example, the Average Precision is $\frac{(1.0+0.67)}{2} = 0.83$. Note how this measure rewards correct suggestions that appear early in the ranking more than those that appear later. For example, if `fGetNearbyObjEq()` had appeared second in the ranking, then the average precision would be $\frac{(1.0+1.0)}{2} = 1.0$. Whereas if `fGetNearbyObjEq()` appeared fourth in the ranking, the Average Precision would be $\frac{(1.0+0.5)}{2} = 0.75$.

Now, suppose Carol's intended query also includes the `RunQA` table, but that SnipSuggest still suggests the recommendations listed above. In this case, the Average Precision drops down to $\frac{(1.0+0.67)}{3} = 0.56$. We divide by 3 (instead of 2) because the number of ground truth features is now 3.

3.6.2 SDSS Dataset

We first evaluate SnipSuggest on the real SDSS dataset.

Quality of Recommendations

We evaluate several aspects of SnipSuggest: its ability to recommend relations, views and tables-valued functions in the `FROM` clause, predicates in the `WHERE` clause, columns and aggregates in the `SELECT` clause, and columns in the `GROUP BY` clause. We evaluate only the SSAccuracy algorithm here because it is able to achieve an interactive response time, and it is the algorithm that aims to solve the Max-Accuracy problem, which corresponds to achieving high average precision. We compare the SSAccuracy and SSCoverage algorithms in the last subsection of Section 3.6.2.

We do a 10-fold cross-validation, with the queries ordered randomly; we use 90% of the queries as the past query workload and test on the remaining 10%. We repeat the experiment 10 times, each time selecting a different set of test queries, so that all queries are part of the test set exactly once. We measure the mean average precision for each of the ten experiments at top-1 through top-10.

Figures 3.5(a) - (c) show the results for predicting snippets in the `FROM` clause. For this experiment, we consider queries with at least three tables, views or table-valued functions in the `FROM` clause. Figure 3.5(a) shows how accurately SnipSuggest recommends snippets, if an empty query is presented. It achieves the same average precision as the Popularity-based algorithm (and thus, SnipSuggest's points are not visible in the graph). This is expected, because with no information, SnipSuggest recommends the most popular snippets across all queries. As soon as the user adds one table to the query (out of three), SnipSuggest's $AP_{@5}$ jumps from 0.339 to 0.77 (nearly a 60% increase). In contrast, the other two techniques' average precisions degrade, because once we add one table to the query, the number of correct snippets has decreased from 3 snippets per query, to only 2 snippets per query. The

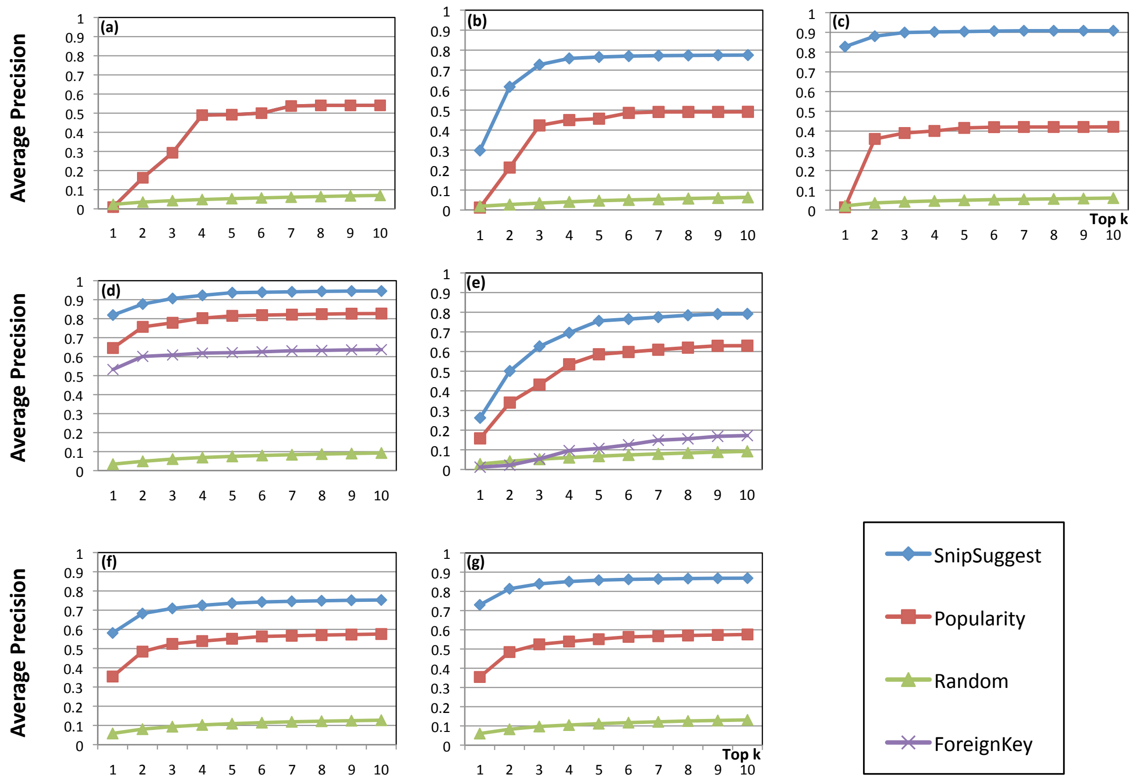


Figure 3.5: Average Precision for recommending tables/views/table-valued functions in `FROM` clause given an empty query, 1 table, or 2 tables (a-c), recommending predicates in `WHERE` clause, for queries with > 0 predicates, or > 1 predicates (d-e), recommending columns in `GROUP BY` clause, given the `FROM` clause, or given both the `FROM` and `WHERE` clauses (f-g). In (a), SnipSuggest's average precision is equal to Popularity's.

Popularity approach's average precision, for example, drops from 0.339 to 0.336. This trend continues when we add two tables to the `FROM` clause. SnipSuggest's average precision jumps to 0.90 (an 84% increase from 0 tables), and Popularity drops to 0.332 (a 16% decrease from 0 tables). In brief, *Figures 3.5(a)-(c) show that SnipSuggest's average precision improves greatly as the user makes progress in writing the SQL query, whereas the other two techniques degrade.*

Figures 3.5(d) - (e) show how accurately SnipSuggest recommends predicates in the `WHERE` clause, for queries with at least one predicate (d), and with at least two predicates (e). In our sample, 75% of the queries have exactly one predicate, and 23% have more. We see that the Popularity approach performs well ($AP_{@5} = 0.81$). This is because all the techniques recommend only valid snippets, and so the Popularity approach restricts its recommended predicates to only those which reference tables that are already in the partial query, and then suggests them in popularity order. Many predicates are join predicates, but the ForeignKey approach still lags because many of the join predicates involve table-valued functions, and thus are not across foreign-key connections. SnipSuggest remains the top approach, achieving an average precision ($AP_{@5}$) of 0.94. Once we consider the less common case, when the query has multiple predicates, Figure 3.5(e) shows a larger difference between the techniques. The ForeignKey technique's performance degrades drastically because we are now looking at mostly non-join predicates. The Popularity approach and SnipSuggest's average precisions also drop (because the queries contain rarer predicates), but now there is a significant discrepancy between the two. In summary, *Figures 3.5(d)-(e) show that both the Popularity and SnipSuggest approaches recommend predicates with high average precision. However, if we consider only queries with multiple predicates, SnipSuggest outperforms the Popularity approach by 29%.*

Figures 3.5(f) - (g) show SnipSuggest's performance for recommending columns in the `GROUP BY` clause, given the `FROM` clause (f) and given both `FROM` and `WHERE` clauses (g). We see a similar trend to recommending snippets in the `FROM` and `WHERE` clauses. SnipSuggest, once again, outperforms the Popularity approach, with $AP_{@5} = 0.86$ versus 0.55. We also see that SnipSuggest's $AP_{@5}$ increases from 0.74 to 0.86 between Figures 3.5(f) and (g). *From Figures 3.5(f)-(g), we learn that, for suggesting snippets in the* `GROUP BY` *clause, Snip-*

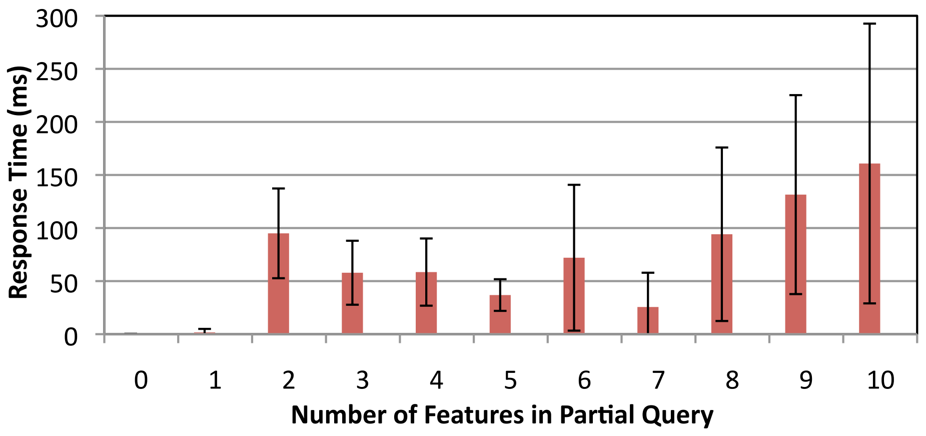


Figure 3.6: Average times to recommend snippets.

Suggest's average precision increases by 16% when the `WHERE` *clause is provided in addition to the* `FROM` *clause, and that SnipSuggest outperforms, by 56%, the Popularity approach.*

For suggesting columns in the `SELECT` clause (not shown), we learn that the Popularity and SnipSuggest approaches perform similarly because most queries select many columns (the average number of columns selected is 12.5), and that the benefit of leveraging the `WHERE` clause, in addition to the `FROM` clause, is small.

Efficiency

Past research shows that a response time of up to 100ms is considered interactive [34]. In the experiments above, SnipSuggest achieves a mean response time of 14ms, and an interactive response time for 94.21% of the partial queries. Figure 3.6 shows the mean response time for different numbers of features in the partial query. The response time can not be determined by the number of features alone; the popularity of the features plays a large role. If the features are popular, there are more relevant queries, and thus more data to process. Therefore, there is no clear trend in the results. We see, however, that the response time increases when we reach 9-10 features. This is because there are often no queries in the workload which contain *all* 9-10 features, and thus SnipSuggest needs to run the SQL in Figure 3.3 multiple times. We exclude partial queries with over 10 features, because there are fewer than 25 such queries in our SDSS query log.

SSAccuracy versus SSCoverage

Now, we compare the SSAccuracy and SSCoverage algorithms. We use a small dataset of only 2000 queries because the SSCoverage algorithm is slow. Since the aim of SSCoverage is different, we use a different measure to evaluate recommendation quality. We define the *utility* of a ranked list of suggestions to be 1 if there is any correct suggestion in the top-k, and 0 otherwise. We report the mean utility across the queries. This is equal to the percentage of queries for which there is a correct suggestion in the top-k.

Figure 3.7 shows the results for predicting columns in the `SELECT` clause, given the `FROM` clause. The difference in the percentage between i and $i+1$ represents the average additional coverage provided by the $i+1$th suggestion. We see that this difference is monotonically decreasing in Figure 3.7, which indicates that SSCoverage suggests the features with the most additional coverage earlier in its ranking. Figure 3.7 shows that *the mean utility of SSCoverage is 15.13% higher than SSAccuracy at the top-5.*

The trend continues for the `FROM` and `WHERE` clauses, though not to the same extent. For the `FROM` clause, given an empty query, SSCoverage achieves a 2% improvement for top-5. Most queries have only one or two tables in the `FROM` clause, so SSAccuracy is guaranteed to suggest features from different queries in the top-5, thus already achieving high coverage. For the `WHERE` clause, SSCoverage outperforms SSAccuracy by 3% in the top-3 (75% of queries contain only one predicate). For queries with multiple predicates, the difference increases to 4%. Although these differences appear small, SSAccuracy already achieves 87% utility at top-3 for the `FROM` clause, and 96% for the `WHERE` clause. Given the little room for improvement, these increases are significant.

3.6.3 IMDB Dataset

We utilize the IMDB dataset for three tasks. First, we study the Query Eliminator's effect on the response time and average precision. Second, we evaluate SnipSuggest over a second dataset. Third, we measure the Query Eliminator's ability to correctly detect end-of-session

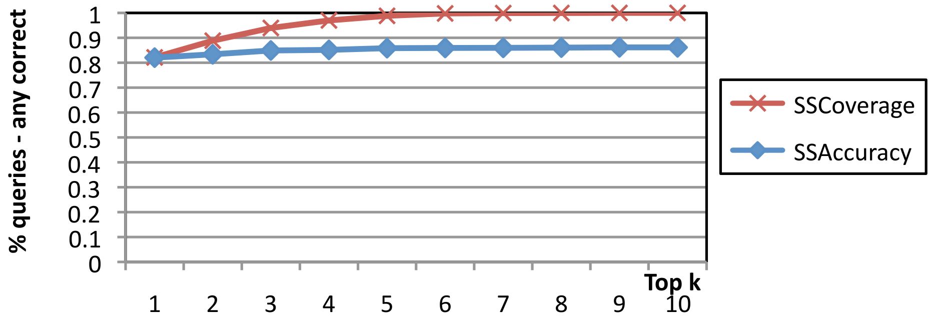


Figure 3.7: SSAccuracy vs. SSCoverage.

Task	Decrease in Time	Increase in $AP_{@3}$
`FROM` $\rightarrow$ `WHERE`	74.49%	8.12%
$\emptyset$ $\rightarrow$ `FROM`	9.17%	4.67%
`1 table in FROM` $\rightarrow$ `FROM`	79.76%	0.74%
`2 tables in FROM` $\rightarrow$ `FROM`	79.47%	-0.71%
`FROM` $\rightarrow$ `SELECT`	88.87%	15.80%
`FROM, WHERE` $\rightarrow$ `SELECT`	79.37%	2.72%
`FROM` $\rightarrow$ `GROUP BY`	77.04%	7.11%
`FROM, WHERE` $\rightarrow$ `SELECT`	60.91%	5.37%

Table 3.2: The benefits and drawbacks of the Query Eliminator.

queries.

Benefits and Drawbacks of Query Eliminator

We summarize the benefits and drawbacks of the Query Eliminator in Table 3.2. From 1679 queries, the Query Eliminator maintains only 7%, or a total of 117 queries. The goal here is to decrease the response time, while maintaining a similar average precision. Table 3.2 shows that the technique decreases the response time by up to 89%. For this dataset, it also increases the average precision ($AP_{@3}$) for all tasks but one. Even in the worst case, the average precision decreases by less than 1%!

Recommendation Quality

First, for SnipSuggest's recommendation quality over this IMDB dataset, we see similar patterns as the SDSS dataset. Namely, the SnipSuggest approach outperforms the other approaches (e.g., by 5 to 20% in average precision in the top-3, in comparison to the Popularity approach), and the recommendation quality improves as the user gives more information (e.g., if there are no tables in the `FROM` clause, SnipSuggest is able to recommend a correct table with 0.62 average precision in the top 2, which increases to 0.91 after a table is added). Although, the general trends still hold, the benefit of the SnipSuggest approach is less significant in the IMDB dataset simply due to the magnitude difference in the schema size (and thus the number of possible features). This schema consists of only five tables and 21 columns.

Query Eliminator Accuracy

We study the session extraction accuracy for only the IMDB dataset. We were able to manually label the session information for this dataset because we have the ground truth in the form of problem numbers from the course assignment. In other words, we can determine which problems (from the assignment) that queries are written for.

Segmentation To quantify the segmentation algorithm's performance, we measure the precision and recall with which it identifies segment boundaries. We compare its performance against using only the time interval between queries. The time interval technique is a common method for extracting sessions from web search logs [31, 48, 49, 92]. The procedure is to set a threshold for the time interval, say 30 minutes, and consider a query to be part of a new session if the time interval between it and the last query is more than the threshold.

Figure 3.8 shows the average precision-recall for the two different segmentation algorithms. We show the average results across 10-fold cross validation. Overall, for SnipSuggest's approach, the area under the curve is 0.930. It is only 0.580 for the time-interval based technique. Our approach significantly outperforms the time-based segmentation technique.

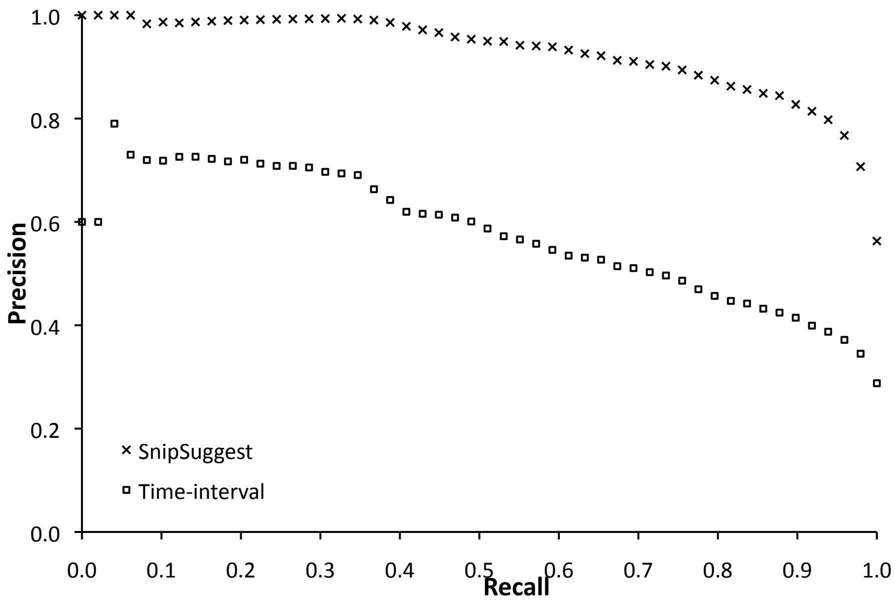


Figure 3.8: Average precision-recall for the Query Eliminator algorithm versus the time interval based algorithm.

Stitching As mentioned above, the most significant segmentation feature for session extraction, is what we call the *AST inclusion type*. This feature represents whether the relationship between the two queries' ASTs is the **Same**, **Add**, **Delete**, **Merge**, **Extract** or **None** (as defined in Table 3.1). We said that two queries are in the same session if this feature has a value of **Same**, **Add**, or **Delete**, with some threshold on the amount of change. We examine how this threshold can affect the performance of the stitching algorithm. Table 3.3 shows the results. μ is the mean difference amount in the training data, among those queries that lie on a session boundary, while σ is the standard deviation. We see that, as expected, smaller thresholds yield better precision while larger thresholds yield better recall. The F-measure, however, remains approximately constant. The mean threshold already recalls 80% of session boundaries, while achieving a precision of 70.7%.

Threshold	μ	$\mu + \sigma$	$\mu + 2\sigma$	∞
Precision	0.707	0.656	0.628	0.578
Recall	0.801	0.893	0.934	1.000
F-measure	0.751	0.756	0.751	0.733
# of Edges	714	896	993	1165

Table 3.3: Precision-recall of stitched edges for different thresholds.

3.7 Conclusion

In this chapter, we presented SnipSuggest, a context-aware, SQL-autocomplete system. SnipSuggest is motivated by the growing population of non-expert database users, who need to perform complex analysis on their large-scale datasets, but have difficulty with SQL. SnipSuggest aims to ease query composition by suggesting relevant SQL snippets, based on what the user has typed so far. We have shown that SnipSuggest is able to make helpful suggestions, at interactive speeds for two different datasets. We view SnipSuggest as an important step toward making query composition easier for both non-expert database users, as well as expert users who are unfamiliar with the database schema.

Chapter 4

PERFXPLAIN: EXPLAINING THE PERFORMANCE OF MAPREDUCE JOBS

Increasingly, users who write MapReduce programs [44, 65], Pig Latin scripts [110], or declarative queries (e.g., HiveQL [72] or SQL) to analyze vast volumes of data are not experts in parallel data processing, but are experts in some other domain. They need to ask a variety of questions on their data and these questions keep changing. For these users to be successful, they need to be self-sufficient in their data analysis endeavors. They cannot rely on administrators or distributed systems experts to help them debug and tune their analysis workloads, because there simply are not enough experts.

While most users already have tools to test and debug the correctness of their SQL queries or MapReduce programs before running them at massive scale, there are limited tools to help understand, diagnose, and debug any performance problems. The performance of parallel programs can be challenging to understand. As an example, when a user runs a MapReduce job and the job seems to take an abnormally long time, the user has no easy way of knowing if the problem is coming from the cluster (e.g., high load or machine failures), from some configuration parameters, from the job itself, or from the input data.

In this chapter, we present PerfXplain, a system that assists users in *debugging the performance of MapReduce applications in a shared-nothing cluster*. PerfXplain lets users formulate performance queries in its own language called the PerfXplain Query Language (PXQL). A PXQL query identifies two MapReduce jobs or tasks. Given the pair of jobs (tasks), the query can inquire about their relative performances: e.g., Why did two MapReduce jobs take the same amount of time even though the second one processed half the data? Why was the last task in a MapReduce job faster than any of the other tasks in that job?

Given a query in PXQL, PerfXplain automatically generates an *explanation* for this query. Informally, an explanation consists of two predicates that hold true about the pair of identified executions. The first predicate, which we refer to as the *despite clause*, maximizes the probability of seeing the expected behavior. Meanwhile, the second predicate, called the *because clause*, maximizes the probability of the observed behavior. For example, if a user asks "why was the last task in this MapReduce job faster than any of the other tasks", an explanation might be: "even though the last task processed the same amount of data as the other tasks (despite clause), it was faster most likely because the overall memory utilization on the machine was lower (because clause) when it executed". When the predicate in the despite clause is true, a pair of tasks typically has the same runtime. *Within that context*, the because clause then explains why the user observed a performance different than anticipated. The despite clause thus helps ensure that the explanation given by the because clause is *relevant* to the identified pair of tasks, rather than just producing a generally-valid argument.

Hence, unlike prior work, which focused on predicting relational query performance [56, 58], predicting MapReduce job performance [55, 102, 103], automatically tuning MapReduce jobs [24, 47, 69, 70, 84] or relational queries [19, 22, 38, 40], and automatically diagnosing failures [50], the goal of PerfXplain is to *explain* the performance similarity or difference between pairs of MapReduce job or task executions. In this project, we focus on explaining runtimes, but our approach can directly be applied to other performance metrics. Additionally, while our implementation and evaluation focus on MapReduce jobs, PerfXplain represents the execution of a single job or task as a vector of features, where each configuration parameter and runtime metric is a feature. As such, the approach is more broadly applicable.

PerfXplain uses machine learning to generate explanations. All performance queries in PerfXplain take the following form: the user specifies what behavior he or she *expected* (e.g., "I expected the last task to take the same amount of time as the others"), optionally why the user expected that behavior (e.g., "all tasks executed the same join algorithm"), and what behavior the user *observed* (e.g., "the last task was faster than the others"). To produce its explanations, PerfXplain utilizes a log of past MapReduce job executions along with their detailed configuration and performance metrics. Given a PXQL query, PerfXplain, identifies positive examples (pairs of jobs/tasks that performed as the user expected), and negative examples (pairs of jobs/tasks that performed as the user observed). From these examples, PerfXplain learns both the most likely reason why the pair should have performed as expected and, within that context, the most likely cause why the pair performed as observed. PerfXplain generates explanations from these two models. The key challenge for generating these explanations is to ensure that every explanation is highly precise, and at the same time as general as possible so that the user can apply this newly acquired knowledge to other scenarios. Overall, we make the following contributions:

1. We propose a simple language, PXQL, for articulating queries about the performance of a pair of MapReduce jobs or tasks (Sections 4.2.1 and 4.2.2).

2. We formally define the notion of a *performance explanation* and three metrics *relevance*, *precision*, and *generality* to assess the quality of an explanation (Section 4.2.3).

3. We develop an approach for *efficiently* extracting performance explanations that have high relevance, high precision, and good generality from a log of past MapReduce job executions (Section 4.3).

4. We evaluate the approach using a log of MapReduce jobs executed on Amazon EC2 [1]. We show that PerfXplain is able to generate explanations with higher precision than two naïve explanation-generation techniques, and offer a better trade-off between precision and generality (Section 4.5).

4.1 Motivation and Overview

We start with a motivating scenario that illustrates the need for PerfXplain. We then present the key types of performance queries that PerfXplain is designed to answer.

4.1.1 PerfXplain Motivation

Parallel data processing systems, such as MapReduce, can exhibit wildly varying performances when executing jobs. Indeed, the performance of a given MapReduce job depends on the following aspects:

1. the details of the computation to perform,

2. the volume of data that must be processed and its characteristics (such as the distribution of values in the input data, which can cause imbalance in processing times between tasks),

3. the current load, hardware configuration, and health of the cluster where the computation is being carried out, and

4. the configuration parameters for the cluster and for the job (block size, number of reducers, amount of memory allocated to the combiner [65], etc.).

Today, it is difficult for users to understand and fix any performance problems associated with their MapReduce computations. Working with scientists at the University of Washington, we have seen numerous cases of these problems. We have even faced such challenges ourselves.

As an example, consider a user who executes a MapReduce job on a 32GB dataset in a cluster with 150 machines. The job takes 30 minutes to run but produces a wrong answer. To debug her job, the user decides to execute it on a smaller, 1GB, dataset. By reducing the size of the dataset, the user hopes to speed-up her debug cycle. However, the smaller dataset also takes 30 minutes to run. Today, the user has limited tools to figure out why both datasets took the same amount of time to process, while the user expected a significant runtime improvement.

PerfXplain's goal is to help users debug this type of performance problem. In this case, the user would pose the following query:

I expected job J_2 to be much faster than job J_1. Why did it take the same amount of time to run?

In this scenario, the explanation is: "because the block size is large". Indeed, because the block size was set to a recommended value of 128 MB, the 32 GB dataset was split into 256 blocks and the 1 GB dataset was split into 8 blocks. Each machine can run two concurrent map and two concurrent reduce tasks (i.e., each machine has two map and two reduce slots), and thus neither the small nor large dataset used the full cluster capacity. The processing time was the time it takes to process one block of data, which is the same for both datasets.

Given such an explanation, the user can then take action. For example, she can reduce the block size or perhaps choose to debug the query locally on the 1GB dataset.

4.1.2 Types of Performance Queries

PerfXplain is designed to answer a variety of queries related to MapReduce application performance. Queries about runtimes refer to two MapReduce jobs or to two MapReduce tasks. The reasoning for this is that a user's expectation for how long a job should take, in general, comes from past experience. This is why we require the user to identify another job as a point of reference. Similarly, tasks have abnormal runtimes only in relation to the runtime of other tasks. By identifying the second job or task, the user clarifies where his runtime expectations come from. We identify two basic types of queries that users may have about the duration of a MapReduce job or task. The first type of queries ask why runtimes were different. The second type asks why runtimes were the same. We illustrate this classification with the following examples:

Example 1 ***Different durations***. *I expected job J_2 to be much faster/slower than job J_1. However, they have almost the same durations. Why?*

Example 2 ***Same durations***. *I expected job J_1 and J_2 to have a similar duration. However, J_2 was much faster/slower than J_1. Why?*

Additionally, performance queries can either be general queries as above or can be constrained queries by the addition of a *despite* clause. Constrained queries can help produce more relevant explanations as we demonstrate in Section 4.5.

Example 3 ***Different durations (constrained)***. *Despite having less input data, job J_2 had the same runtime as J_1. I expected J_2 to be much faster. What is the explanation?*

Example 4 ***Same durations (constrained)****. Despite having a similar input data size and both using the same number of instances, J_2 was much slower than J_1. I expected both to have a similar duration. What is the explanation?*

Finally, similar types of queries can be asked both for jobs and for tasks. Task runtimes can be compared within and across jobs.

Example 5 *I expected all map tasks to have similar durations since they processed the same amount of data. However, task T_2 was faster than the other tasks, e.g., T_1. Why was this the case?*

4.2 Performance Queries

We introduce PerfXplain's data model and language.

4.2.1 Data Model

Job and Task Representation: To generate its explanations, PerfXplain assumes that it has access to a log of past MapReduce job executions. PerfXplain models job executions using the following schema for jobs:

$$\texttt{Job(JobID,feature}_1\texttt{,...,feature}_k\texttt{,duration)}$$

and the following schema for MapReduce tasks:

$$\texttt{Task(TaskID,JobID,feature}_1\texttt{,...,feature}_l\texttt{,duration).}$$

The features for MapReduce jobs include configuration parameters (e.g., DFS block size, number of reduce tasks), system performance metrics (e.g., metrics collected by Ganglia [4]), data characteristics (e.g., input data size), and application-level details (e.g., the relational operator corresponding to the MapReduce job if the job was generated from HiveQL or Pig Latin). In our current implementation, the features for tasks include all features that are collected in the MapReduce log files (e.g. the task type, map input bytes, map output bytes), the MapReduce job it belongs to, as well as all the system performance metrics collected by Ganglia during the task execution. PerfXplain comes configured with collecting these specific features but can easily be extended to use additional features.

Throughout the chapter, we will refer to job and task executions with their `JobID` or `TaskID`, respectively. To refer to the value of a feature `f` for a specific job `J`, we will use the notation `J.f`.

Representation of Examples: Because PerfXplain answers queries about *pairs* of jobs (or tasks), all the examples that it learns from come in the form of pairs of jobs. We refer to a pair of jobs as a *training example*. A training example consists of $4 \cdot k$ features, where k is the number of features that we collect for a single job or task (we call these the *raw features*).

Table 4.1 lists the features that we compute for each training example. The left column enumerates the set of features (which we refer to as `F`), and the right column specifies the domain for each feature. We assume that we know the domains of the raw features. We denote the domain of a feature `f` with `dom(f)`.

The computed features (i.e., those listed in Table 4.1) encode the relationship between the two jobs for each raw feature, at varying levels of resolution. The first set of features, which are of the form $\texttt{f}_i\texttt{\_isSame}$, are binary features that represent whether the two jobs have the same value for $\texttt{feature}_i$. The second set of features, of the form $\texttt{f}_i\texttt{\_compare}$, represent whether J_1's value for $\texttt{feature}_i$ is much less than (`LT`), similar to (`SIM`) [1], or much greater than (`GT`) $\texttt{J}_2$'s value for $\texttt{feature}_i$. This feature is appropriate only for numeric features and thus the value of the feature is set to be missing for nominal features. Similarly, the third set of features, which are of the form $\texttt{f}_i\texttt{\_diff}$ represent the change in value for $\texttt{feature}_i$. This feature is computed only for nominal features, and is thus set to be missing if a feature is numeric. For example, if the value for `pigscript` for $\texttt{J}_1$ is `filter.pig` and for $\texttt{J}_2$ is `join.pig`, then the value of `pigscript_diff` is `(filter.pig, join.pig)`. We refer to these three sets of features as *comparison features*, because they compare the raw features of the two jobs

[1]In the current implementation, two values are considered to be similar if they are within 10% of one another.

(or tasks). Finally, the fourth set of features are directly copied from the jobs if the jobs have the same value for that feature. Namely, feature $\texttt{f}_i$ is set to the value $\texttt{J}_1\texttt{.feature}_i$ if $\texttt{J}_1\texttt{.feature}_i = \texttt{J}_2\texttt{.feature}_i$. Otherwise, the feature is labeled as missing. We refer to these features as the *base features*.

The key intuition behind the above feature choice is that they span the range from general features (i.e., `_isSame` features) to specific features (i.e., base features). The general features help abstract details when they are not important, which has two implications. First, explanations can become more generally applicable. Second, pairs of jobs that have very different raw features can become comparable. For example, if a task had a different runtime than another because the load on the instance was different, PerfXplain can generate an explanation of the form "CPU utilization isSame = false" rather than "CPU utilization when running task 1 was X while CPU utilization when running task 2 was Y". At the same time, detailed features are sometimes needed to get precise explanations when details matter. For example, the reason why a job took the same amount of time as another even though it used more instances could be "because the block size was larger than or equal to 128MB".

4.2.2 PXQL Syntax

PXQL allows users to formulate queries over the performance of either MapReduce jobs or tasks. To simplify the presentation, we focus only on jobs in this section.

A PXQL query consists of a pair of jobs and three predicates over their features. The first two predicates describe the observed behavior for the two jobs and the reason why the user is surprised by this behavior. The third predicate specifies what behavior the user expected. Every predicate takes the form $\phi_1 \wedge \ldots \wedge \phi_m$, where each ϕ_i is of the form f `op` c where f is a feature from Table 4.1, c is a constant, and `op` is an operator. The set of operators supported by PerfXplain include $=, \neq, <, \leq, >$ and , $\geq$.

Definition 13 *A* **PXQL query Q** *comprises a pair of jobs* (J_1, J_2) *and a triple of predicates* $(\mathbf{des}, \mathbf{obs}, \mathbf{exp})$*, where* **des***,* **obs** *and* **exp** *are predicates over* J_1 *and* J_2*'s features.*

Feature	Domain
$\texttt{f}_1\texttt{\_isSame}$	{T, F}
...	
$\texttt{f}_k\texttt{\_isSame}$	{T, F}
$\texttt{f}_1\texttt{\_compare}$	{LT, SIM, GT}
...	
$\texttt{f}_k\texttt{\_compare}$	{LT, SIM, GT}
$\texttt{f}_1\texttt{\_diff}$	$\texttt{dom(feature}_1\texttt{)} \times \texttt{dom(feature}_1\texttt{)}$
...	
$\texttt{f}_k\texttt{\_diff}$	$\texttt{dom(feature}_k\texttt{)} \times \texttt{dom(feature}_k\texttt{)}$
$\texttt{f}_1$	$\texttt{dom(feature}_1\texttt{)}$
...	
$\texttt{f}_k$	$\texttt{dom(feature}_k\texttt{)}$

Table 4.1: Set of features that define a training example. The features are computed for a pair of jobs (tasks), and encode the relationship between the two jobs (tasks) for each raw feature, at varying levels of resolution.

Additionally, $\mathbf{des}(J_1, J_2) = \text{true}$*,* $\mathbf{obs}(J_1, J_2) = \text{true}$*, but* $\mathbf{exp}(J_1, J_2) = \text{false}$*. Furthermore, it must be the case that* $\mathbf{obs} \models \neg\mathbf{exp}$*.*

We refer to (J_1, J_2) *as the* **pair of interest***, and the predicates as the* **despite***,* **observed***, and* **expected** *clauses, respectively.*

We use the following syntax for PXQL queries.

```
FOR J1, J2 WHERE J1.JobID = ? and J2.JOBID = ?
DESPITE des OBSERVED obs
EXPECTED exp
```

Informally, a PXQL query $Q = (\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ over the pair of jobs J_1, J_2 can be read as "Given jobs J_1 and J_2, despite **des**, I observed **obs**. I expected **exp**. Why?" PerfXplain's goal is to then reply with an explanation of the form:

1. `OBSERVED` $duration\_compare = SIM$

   `EXPECTED` $duration\_compare = GT$

2. `OBSERVED` $duration\_compare = LT$

   `EXPECTED` $duration\_compare = SIM$

3. `DESPITE` $inputsize\_compare = GT$

   `OBSERVED` $duration\_compare = SIM$

   `EXPECTED` $duration\_compare = GT$

4. `DESPITE` $inputsize\_compare = SIM \ \wedge$

   $numinstances\_isSame = T$

   `OBSERVED` $duration\_compare = LT$

   `EXPECTED` $duration\_compare = SIM$

5. `DESPITE` $inputsize\_compare = SIM \ \wedge$

   $jobID\_isSame = T$

   `OBSERVED` $duration\_compare = GT$

   `EXPECTED` $duration\_compare = SIM$

Figure 4.1: Example PXQL queries.

`DESPITE` **des′**

`BECAUSE` **bec**

where **des′** is a an extension of the user's **despite** clause and **bec** is a predicate over the features of the MapReduce jobs that appeared in the query.

Figure 4.1 shows how each example from Section 4.1 translates into a PXQL query. We omit the FOR clause. For example, the first query asks why the two jobs had a similar duration (`duration_compare = SIM`) and that the user expected that J_1 would be slower than J_2 (`duration_compare = GT`). As illustrated in the first two examples, the **despite** clause is optional. Omitting the clause is equivalent to setting **des** to *true*. Example 5 shows that the same query language that we use for jobs serves to ask performance queries over tasks.

4.2.3 PXQL Semantics

Given a PXQL query, PerfXplain must present the user with an explanation.

Definition 14 *For a query* $Q = (\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ *over a pair of jobs* (J_1, J_2)*, a* **candidate explanation** E *is a pair of predicates* $(\mathbf{des}', \mathbf{bec})$*. The predicates are referred to as the* **despite** *and* **because** *clauses, respectively.*

For instance, for Example 1, a candidate explanation is $E = (\mathbf{des}', \mathbf{bec})$, where $\mathbf{des}' =$ ($\texttt{inputsize\_compare} = \texttt{GT}$) and $\mathbf{bec} = (\texttt{blocksize} >= \texttt{128MB} \wedge \texttt{numinstances} \geq \texttt{100})$.

The first requirement from an explanation is that it holds true for the pair of jobs that the user is asking about. For example, explanation E above says that the reason why the durations of J_1 and J_2 were similar is because the two jobs both had a large block size and a large number of instances. However, this explanation would not make sense if J_1 and J_2 did not satisfy these conditions. In such a case, we say that E is not applicable to (J_1, J_2).

Definition 15 *A candidate explanation* $E = (\mathbf{des}', \mathbf{bec})$ *is* **applicable** *to a pair of jobs* (J_1, J_2) *if* $\mathbf{des}'(J_1, J_2) = \text{true}$ *and* $\mathbf{bec}(J_1, J_2) = \text{true}$.

The applicability requirement for an explanation is a hard requirement. Every explanation generated by PerfXplain *must* be applicable. Additionally, we define three metrics of the quality of an explanation for a given log of MapReduce job executions.

Definition 16 *The* ***relevance****,* $Rel(E)$*, of an explanation* $E = (\mathbf{des}', \mathbf{bec})$ *given a PXQL query* $(\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ *is the following conditional probability:*

$$Rel(E) = P(\mathbf{exp} | \mathbf{des}' \wedge \mathbf{des}). \tag{4.1}$$

Intuitively, an explanation with high relevance identifies (through the $\mathbf{des}' \wedge \mathbf{des}$ clause) the key reasons why the pair of jobs *should have* performed as expected. For example, if we consider our explanation E from above, it has a high relevance because its **des** clause specifies that it consider only pairs of jobs where `inputsize_compare = GT`. Indeed, given that the input size of J_1 is greater than J_2, we would expect that J_1 be slower than J_2. By considering only pairs of jobs that satisfy the $\mathbf{des}' \wedge \mathbf{des}$ clause, the explanation given by the **bec** clause is more relevant because it focuses on circumstances that are specific to the user query. In our example, the **bec** clause identifies why pairs of jobs where one job consumes a much greater input still can have the same runtime. This explanation is more relevant to the query than one which would have explained why a job can have the same runtime as another job, *in general.*

Definition 17 *The* ***precision****, $Pr(E)$, of an explanation $E = (\mathbf{des}', \mathbf{bec})$ given a PXQL query $(\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ is the following conditional probability:*

$$Pr(E) = P(\mathbf{obs}|\mathbf{bec} \wedge \mathbf{des}' \wedge \mathbf{des}). \tag{4.2}$$

A precise explanation tries to identify why, in the context of $\mathbf{des}'$ and **des**, did the pair in question most likely perform as it did instead of as expected. For example, consider $E' = (\mathbf{des}', \mathbf{bec}')$, where $\mathbf{bec}' =$ `blocksize` $>=$ `128MB`. This is a shorter version of E from above. E' has most likely a lower precision than E because it is rarely the case that two jobs have a similar runtime just because they have the same large block size. On the other hand, if the two jobs also executed in a large cluster, then it is likely that neither used the full cluster capacity and the runtime was determined by the time to process one large block of data.

Though precision is necessary, an explanation with high precision may still be undesirable. Consider the following **because** clause: `start_time = 1323158533` $\wedge$ `instance_url = 12-31-39-E6.compute-1.internal:localhost/127.0.0.1`. Such an explanation can have a precision of 1.0, yet it is still not a good explanation. A good explanation is one that can apply to more than one setting. In fact, we posit that the more settings where an explanation applies, the better the explanation, because it identifies more general patterns in job performance. We measure this third property with the following metric.

Definition 18 *The **generality**, $Gen(E)$, of an explanation $E = (\mathbf{des}', \mathbf{bec})$ given a PXQL query $(\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ is the following conditional probability:*

$$Gen(E) = P(\mathbf{bec} | \mathbf{des}' \wedge \mathbf{des}). \tag{4.3}$$

Note that precision and generality are closely related to the data mining concepts of confidence and support, respectively. The only difference is that our terms explicitly refer to the various clauses of the explanation. Namely, precision is the confidence that the **because** clause leads to observed behavior in the context of the **despite** clause, and generality is the support of the **because** clause in the context of the **despite** clause.

Given a PXQL query, PerfXplain's goal is to generate an applicable explanation that achieves high precision, relevance and generality. However, as in the example above, precision and generality are usually in direct conflict with one another. Thus, a helpful explanation must strike a good balance between the two metrics.

Finally, PerfXplain orders the predicates in the **despite** and **because** clauses so that the important predicates appear first. A predicate is more important than another if it achieves higher marginal relevance (in the **despite** clause) or higher marginal precision (in the **because** clause).

4.3 PXQL Query Evaluation

In this section, we describe how PerfXplain generates explanations for PXQL queries. We begin with a few definitions.

4.3.1 Terminology

Given a PXQL query and a pair of jobs in the log, we first say that the pair of jobs is *related* to a query if it satisfies the **des** clause and either the **expected** or **observed** clauses.

Definition 19 *A pair of jobs (J_1, J_2) is* **related** *to a PXQL query $Q = (\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ if $\mathbf{des}(J_1, J_2) = \text{true} \wedge (\mathbf{exp}(J_i, J_j) = true \vee \mathbf{obs}(J_i, J_j) = true)$.*

Further, we say that a related pair of jobs performed as expected or as observed with respect to the query depending on whether it satisfied the **expected** or **observed** clause.

More formally:

Definition 20 *A pair of jobs* (J_i, J_j) **performed as expected** *with respect to a PXQL query* $Q = (\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ *if* $\mathbf{des}(J_i, J_j) = \text{true} \wedge \mathbf{exp}(J_i, J_j) = \mathit{true}$.

Similarly,

Definition 21 *A pair of jobs* (J_i, J_j) **performed as observed** *with respect to a PXQL query* $Q = (\mathbf{des}, \mathbf{obs}, \mathbf{exp})$ *if* $\mathbf{des}(J_i, J_j) = \text{true} \wedge \mathbf{obs}(J_i, J_j) = \mathit{true}$.

4.3.2 Approach

Given a query Q, PerfXplain generates an explanation in the form of a pair of **des**' and **bec** clauses. The constructions of these two clauses is symmetrical. We first explain how PerfXplain generates the **bec** clause.

Overview of bec clause generation. The **bec** clause generation takes two inputs. The first input is the log of past MapReduce job executions. Each pair of jobs in the log forms a training example, which is represented by a combined vector of features as shown in Table 4.1. These job pairs and their features serve as the basis for generating the explanation. The second input is the PXQL query itself. The query comprises the pair of jobs of interest, (J_1, J_2) and the three predicates: $(\mathbf{des}, \mathbf{obs}, \mathbf{exp})$.

The key idea behind performance explanation is to identify the conditions why the pair of interest performed as observed rather than performing as expected. This condition takes the form of a predicate on the job-pair features (i.e., those listed in Table 4.1). As discussed in the previous section, we want an explanation that is both precise and general: an explanation is precise if whenever a pair of jobs satisfies it, that pair is likely to perform as observed. At the same time, an explanation is general if it applies to many pairs of jobs in the log.

Detailed algorithm for bec clause generation. Algorithm 2 shows the detailed **bec** clause generation approach. The algorithm takes as input a PXQL query, the pair of interest (J_1, J_2), the set of all jobs J, and the desired explanation width w. The width is the number of atomic predicates in the explanation.

Lines 1-2: Construct training examples. The first step in the explanation generation process (i.e., `constructTrainingExamples`) identifies the related pairs of jobs in the log. Only pairs that satisfy the **des** predicate and either the **obs** or **exp** predicates are used to generate an explanation for the given query. The **obs** and **exp** predicates also serve to classify job pairs as performing either as observed or as expected. Next, the algorithm keeps just a sample of this set. We further discuss sampling in Section 4.3.3.

Given these training examples, the algorithm generates the explanation as a conjunction of atomic predicates. It grows the explanation by adding atomic predicates in a greedy fashion. To select each atomic predicate, the algorithm identifies (a) the "best" predicate for each feature, and then (b) selects the "best" predicate across features.

Line 5: Construct best predicate for each feature. An atomic predicate is of the form f `op` c. Thus, given a feature f, in order to find the best predicate, PerfXplain must select the best `op` and constant c pair. For nominal attributes, the only operator it considers is equality. For numeric attributes, it considers both equality and inequality operators.

In order to select the best predicate for a feature, PerfXplain identifies the predicate with the highest information gain, which is defined as:

$$Information\_Gain(P, \phi) = H(P) - H(P|\phi)$$

where ϕ is the predicate, P is the pairs of jobs in consideration, and $H(P)$ is the information entropy of P. When we consider ϕ, we think about the two partitions that ϕ creates: the pairs that satisfy ϕ and the pairs that do not. By maximizing the information gain, we want to find the predicate that leads to two partitions where the partitions each have a lower entropy (or higher "purity") than the entropy of the full set of pairs.

As an example, consider Figure 4.2. The leftmost box represents the full set of training examples we are considering. The training examples are depicted with either a $+$ (if an example performed as observed) or with a $-$ (if it performed as expected). Suppose for a feature f, we are considering two predicates. For example, for `blocksize` we may be considering `blocksize > 64MB` and `blocksize` $\leq$ `256MB`. The two predicates are illustrated by the second (A) and third (B) boxes in Figure 4.2. The grey area represents where a

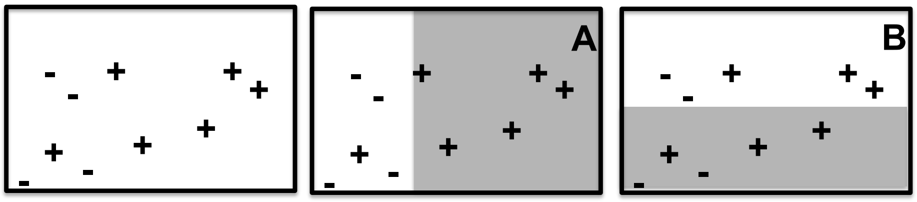


Figure 4.2: Example of information gain. The training examples are depicted with either a $+$ or a $-$. The leftmost box represents the full set of training examples. A achieves higher information gain than B.

training example satisfies the predicate. Now, if we consider the two predicates, clearly A is a better predicate than B, because it is doing a better job of separating the $+$'s from the $-$'s. The information gain metric captures exactly this intuition. Entropy is defined as $H(P) = -plog_2(p) - (1-p)log_2(1-p)$ where p is the fraction of $+$'s. In our example, $p = 0.6$ for the full sample. Thus, our original entropy is 0.97. The entropy of the sample in A is 0.1, which is calculated by taking the weighted average of the entropies of both the grey side and the white side (the two partitions created by the predicate). Thus, the information gain for A is 0.87. For B, the entropy is 0.97, which is an information gain of 0. Therefore, the predicate depicted in A is better than the predicate depicted in B.

Lines 6-15: Identify the best cross-feature predicate. For each of the above per-feature predicates, the algorithm computes its precision and generality. Both precision and generality are measured over the set of job-pairs (P in Algorithm 2) that are related to the PXQL query and satisfy the explanation constructed so far. We compute precision as the number of jobs-pairs that satisfy the predicate and perform as observed, divided by the number of job-pairs in that satisfy the predicate. Generality is the fraction that satisfies the predicate. The score of a predicate then becomes a weighted average of its precision and its generality scores (line 13). In the current implementation, we use a weight of $w = 0.8$ (thus favoring precision over generality).

Note, however, that the score is not simply a weighted average over the raw precision and generality scores. Instead, it calculates a relative score for each. Consider the precision score, or *precScore*, as an example. To calculate it, PerfXplain computes the precisions of all the predicates, ranks them, and replaces the precision values with the percentile ranks. PerfXplain does the same transformation for the generality score. In our earlier implementation, we had not included this step, and we found that because the generality scores tended to be much lower than the precision scores (especially as the explanation grew in width), the generality was not having enough impact on the predicate score. Therefore, we introduced this step to normalize the two scores before taking their weighted average. Finally, the predicate with the highest score is added to the explanation.

Lines 16-18: Extend explanation and continue. To further refine the explanation, PerfXplain iterates and adds additional atomic predicates. At each iteration, PerfXplain considers only those job pairs that satisfy the **bec** predicate generated so far. Some of the job pairs still performed as expected in that set. PerfXplain then identifies an additional atomic predicate that isolates these job pairs from the others while correctly classifying the pair of interest. The resulting extended predicate forms a more precise explanation for why the pair of jobs performed as observed rather than performing as expected. The algorithm stops once a clause of width w has been generated.

The result of the algorithm is an explanation consisting of a **bec** clause with w atomic predicates.

Generating the des' clause. Given a PXQL query, the **bec** predicate strives to capture a precise yet general reason why some jobs performed as observed rather than performing as expected. The **bec** predicate is restricted to hold for the pair of interest identified in the query. In spite of this constraint, we found that it was often the case that the explanation would produce *overly generic* reasons why a pair of jobs performed as observed rather than performing as expected. For example, consider the case where a user asks why two jobs had the same runtime instead of one job being faster than the other. In the absence of a **des** clause, a general and precise explanation of width 1 says that the two jobs executed on the same number of instances. Instead, if the system generates the explanation not by using the entire log but by only considering the subset of job pairs where one job processed a significantly larger amount of data than the other, the most precise and general explanation changes. For this subset of jobs, the explanation becomes about block size and cluster size. The latter explanation is more relevant to the pair of interest. The **des** clause captures this intuition in a principled fashion.

In the current implementation, by default, PerfXplain generates only the **bec** clause in an explanation, and the user must explicitly request a **des** clause. An easy modification is to set a relevance threshold r. If the user's **des** clause achieves a relevance score less than r, then PerfXplain will extend the clause automatically until its score reaches this threshold or it can not further be improved.

Conveniently, the **des**' clause generation technique is symmetric to the **bec** clause generation. In order to generate the **des**' clause, PerfXplain uses the same algorithm as shown in Algorithm 2. However, it changes line 6 to measure relevance $P(\mathbf{exp}|p)$ instead of precision $P(\mathbf{obs}|p)$.

Once PerfXplain has generated a sufficient **des**' clause, PerfXplain verifies the clause with the user. If the user approves this clause, it is added to the user's PXQL query, and PerfXplain can proceed to generating the **bec** clause, and thus a full explanation.

Comparison to other machine-learning techniques. Explanation generation is related to classification problems in machine learning. In particular, our approach is related to decision trees [113] since both identify predicates over features that separate examples into two classes (observed and expected in our case). There are however several important distinctions. First, unlike a decision tree, performance explanation must ensure that the pair of interest is always correctly classified as performing as observed. Second, performance explanation need not categorize all pairs of jobs in the log. Instead, it must generate a predicate that yields a relevant, precise, and general explanation *given the pair of interest.* In order to achieve this goal, performance explanation must construct a **des**' clause *before* generating the **bec** clause. Additionally, it must consider the precision and generality metrics during the construction of each of these two clauses.

While we cannot apply decision trees directly to the performance explanation problem, we still re-use the notion of information gain for constructing the best predicate for each feature. In our prototype, we use the C4.5 [113] technique for finding the predicate that maximizes the information gain metric.

4.3.3 Sampling

To maintain a low response-time for the explanation generation, PerfXplain samples the training examples related to the current query (line 2 of Algorithm 2). Sampling also helps balance the number of positive and negative examples that will contribute to the explanation. A balanced sample is one in which there is approximately the same number of examples labeled as *observed* and as *expected*. A highly unbalanced sample can cause PerfXplain to believe that a trivial explanation is sufficient. For example, if a sample consists of pairs where 99% performed as observed, PerfXplain will decide that the empty explanation is good as it will achieve a precision of 99%.

The sampling method operates as follows. It iterates through each training example. If the desired sample size is m and T is the set of all training examples, then the sampling technique keeps a training example t with probability:

$$p = \begin{cases} m/(2 \cdot |\{x \in T : \mathbf{obs}(x) = true\}|) & \text{if } \mathbf{obs}(t) = true \\ m/(2 \cdot |\{x \in T : \mathbf{exp}(x) = true\}|) & \text{if } \mathbf{exp}(t) = true \end{cases}$$

For instance, consider a set of training examples that consists mostly of pairs labeled with *observed* and very few with *expected*. In this case, a training example labeled with *observed* will have a lower probability of being selected for the sample than a training example labeled with *expected*. In our current implementation, we use a sample size of 2000.

Currently, PerfXplain randomly samples training examples, which already yields high-quality explanations as we show in Section 4.5. Biasing the samples in some way, such as ensuring that priority is given to executions that correspond to a varied set of jobs, could possibly improve explanation quality further. We leave this question open for future work.

4.4 Alternative Approaches

In this section, we describe two naïve techniques for constructing explanations. Though at first glance, both techniques seem like they may be sufficient for generating good explanations, we see that both fall short in different ways. We compare the PerfXplain approach to these techniques in Section 4.5.

4.4.1 RuleOfThumb Approach

This technique first identifies which features of a job have a high impact on the runtime of a job in general. Then it points to differences in these features as the explanation. This identification of important features is executed only once, and is not performed per PXQL query. Once the user issues a PXQL query along with a pair of interest (J_1, J_2), RuleOfThumb returns the top-w features that the two jobs disagree on (where w is the width of the explanation desired). The features are ranked by their importance as determined in the step described above.

Consider the following example. Suppose that the initial stage identifies that `numinstances`, `inputsize`, and `num_reduce_tasks` are the most important features for determining the duration of a job, respectively. Suppose the user has asked why job J_1 is slower than job J_2, and that both jobs agree on the number of instances, but disagree on the input data size and on the number of reduce tasks. In this case, the explanation generated would be:

`inputsize_isSame = F` $\wedge$ `num_reduce_tasks_isSame = F`.

Any standard feature selection algorithm can be used to determine the most important features. In our implementation, we use the Relief technique [114] because it is able to handle both numeric and nominal attributes, as well as missing values.

The RuleOfThumb algorithm works well for some PXQL queries. For example, it may be appropriate for queries that ask for an explanation of why the runtime of two jobs is different because the technique always points to differences in important features and differences in features usually lead to differences in the runtime. However, this approach completely ignores the PXQL query, and will therefore, fail to satisfactorily answer many queries.

4.4.2 SimButDiff Approach

Unlike the previous technique, the SimButDiff algorithm actually considers the PXQL query when generating its explanation. It first finds all training examples that are similar to the pair of interest, with respect to its `_isSame` features. Among these similar pairs, for each feature f_i, it measures the fraction of pairs that performed as expected and disagreed on this feature to the number of pairs that disagreed on this feature. In essence, it performs 'what-if' analysis on each feature f_i to check the following: if this feature had been different, how likely is it that the pair would have performed as expected. For example, if the pair of interest agree on `numinstances`, it finds all pairs that were similar to the pair of interest, but disagreed on the `numinstances` to see if that generally leads to pairs that performed as expected. It measures this fraction for each feature, and the features that have the highest fractions constitute the explanation.

Algorithm 3 shows the details of this approach. In addition to the PXQL query, the pair of interest, the set of all jobs, and the desired width, the algorithm also takes as input a similarity threshold s between 0 and 1. In the current implementation, a similarity threshold of 0.9 has worked well.

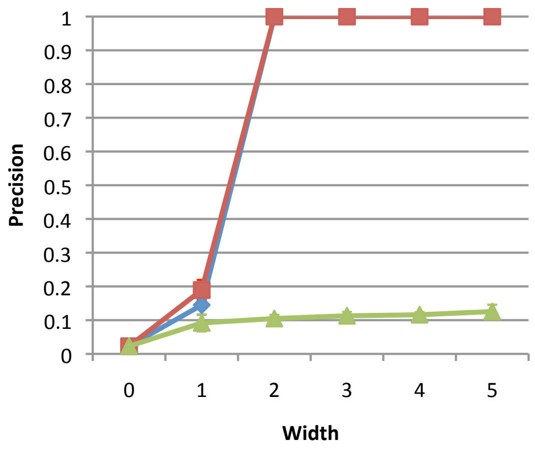
The algorithm proceeds as follows. First, like the PerfXplain algorithm, it constructs the training examples (line 1). However, it keeps only the `_isSame` features (lines 2-3). Next, it filters out training examples that are not similar to (J_1, J_2), the pair of interest (lines 4-5). A training example is similar if it agrees with the pair of interest on at least s fraction of the `_isSame` features.

Next, the algorithm iterates through every feature and calculates a score for it (lines 6-11). The score for a feature f is the fraction of training examples that perform as expected among those that *disagree* with (J_1, J_2) on f. The features are then sorted in descending order of these scores (line 12) and the explanation is a conjunction of predicates of the form $f = (J_1, J_2).f$, constructed in order of the score (lines 14-17).

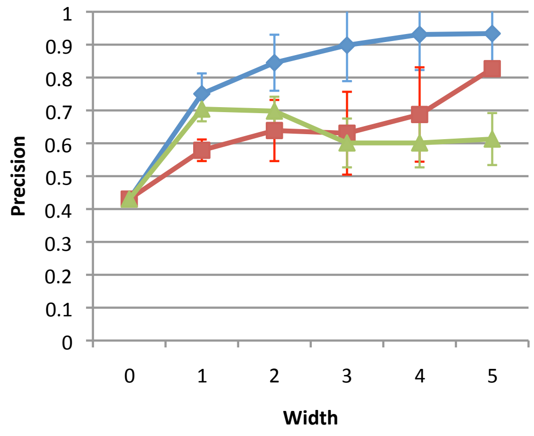
The SimButDiff algorithm only utilizes the `_isSame` features because they are binary features, and thus it is easy to identify the training examples that disagree with (J_1, J_2) on a feature (i.e., check that the training example takes on the one different value). Secondly, it is simple to measure the similarity of two training examples by just counting the number of features that they agree on. Were we to leverage some of the other features, we would need to define similarity scores between the different values in the domain of the feature.

As such, because the SimButDiff technique leverages only the `_isSame` features, it fails to produce precise explanations where more complex features are required. We show examples of such PXQL queries in Section 4.5.

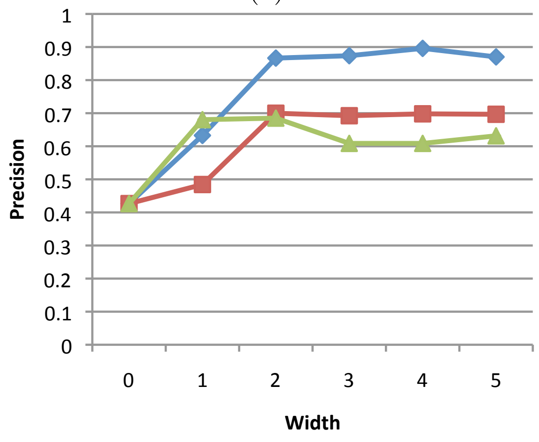
4.5 Evaluation



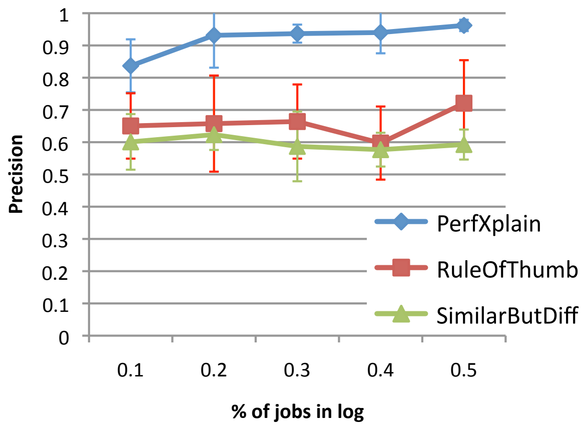
(a)



(b)



(c)



(d)

Figure 4.3: Explanation precisions for (a) **WhyLastTaskFaster**, and (b) **WhySlowerDespiteSameNumInstances** with varying width. Precisions for (c) when the log consists only of simple-groupby.pig jobs, and (d) at width-3 with varying log size for **WhySlowerDespiteSameNumInstances**.

In this section, we compare the three explanation generation approaches on two PXQL queries over real data that we collected on Amazon EC2. We explore several aspects of the techniques. In Section 4.5.3, we assume that the user has given us a well-specified PXQL query, and we compare the precisions of the explanations generated by each technique. In Section 4.5.4, we assume that the user has provided an under-specified query, and investigate

Parameter	Different Values
Number of instances	1, 2, 4, 8, 16
Input file size	1.3 GB, 2.6 GB
DFS block size	64 MB, 256 MB, 1024 MB
Reduce tasks factor	1.0, 1.5, 2.0
IO sort factor	10, 50, 100
Pig script	simple-filter.pig, simple-groupby.pig

Table 4.2: The parameters we varied and the different values for each.

whether PerfXplain is able to generate an effective **despite** clause. In Section 4.5.5, we explore the case where the log consists of only one type of MapReduce job, whereas the pair of jobs in question are of a different type. We evaluate the impact of the log size on the precision of explanations in Section 4.5.6. Finally, in Section 4.5.7, we analyze the trade-off between generality and precision. We begin with a description of the experimental setup (Section 4.5.1) and the two PXQL queries that we use for this evaluation (Section 4.5.2).

4.5.1 Experimental Setup

To collect real data for our experiments, we ran two different Pig scripts on Amazon EC2 and varied several parameters for each execution. Table 4.2 shows the features that we varied and the different values that we tried for each one.

The number of instances is the number of virtual machines used for the job. The input file consists of a log of search queries submitted to the Excite [53] search engine. This is a sample file that is used in the standard Pig tutorial. We concatenate the sample file from the tutorial to itself either 30 or 60 times. This input data file is then broken up into a given number of blocks. The block size is set through the `dfs.block.size` parameter in the MapReduce configuration file. The block size determines the number of map tasks that are generated for an input file (i.e., the number of map tasks is the input file size divided by the block size). The reduce tasks factor determines the number of reduce tasks for the job, using the `mapred.reduce.tasks` parameter. The factor is in relation to the number of instances. e.g.. If there are 8 instances, and the reduce tasks factor is 1.5, then the number of reduce tasks is set to 12. The IO sort factor feature corresponds to the `io.sort.factor` MapReduce parameter, and represents the number of segments on disk to be merged together at a given time. Finally, the Pig script parameter specifies which Pig job should be executed. The `simple-filter.pig` script simply loads the input file, filters out all queries where the query string is a URL, and outputs the queries that remain. The `simple-groupby.pig` script groups all the queries by user and outputs the number of queries per user.

PerfXplain collects data at the job-level as well as the task-level. For each task, PerfXplain extracts all details it can from the MapReduce log file, including `hdfs_bytes_written`, `hdfs_bytes_read`, `sorttime`, `shuffletime`, `taskfinishtime`, and `tracker_name`. We refer the reader to a Hadoop guide for details of these properties [127]. Additionally, PerfXplain also monitors the instances using Ganglia [4], which is a distributed monitoring system. Ganglia metrics include `boottime`, `bytes_in`, `bytes_out`, `cpu_idle`, and more. To be precise, PerfXplain runs Ganglia to measure these metrics on each instance once every five seconds. For a given task, it identifies the instance that the task was executed on, and for each metric, it calculates the average value while the task was executing. PerfXplain also percolates this monitoring data up to the jobs. Namely, for each job and each metric, it calculates the average value of the metric across all the tasks belonging to the job. In total, PerfXplain records a total of 64 features for each task and 36 features for each job.

The graphs in this section are generated as follows. We divide the log into two logs: the *training log* and the *test log*. This split is done randomly; we iterate through each job, add it to the training log with 50% probability, and all remaining jobs are added to the test log. The training log is used as the basis for generating the explanation. The test log is used to evaluate the explanation. Namely, we measure the precision of the explanation over the test log. We repeat this process ten times, and our graphs report the average results across these ten runs, along with errors bars to depict the standard deviation.[2]

[2]A common evaluation method used in machine learning literature is ten-fold cross validation. We did not use this technique because it leads to a small test log consisting of a tenth of the jobs. A small log is ineffective for testing because it results in too few pairs that performed as observed. Therefore, we used 2-fold cross validation.

4.5.2 The PXQL Queries

We evaluate how well PerfXplain generates explanations for two different PXQL queries. The first query asks why one task is faster than another, and the second asks why one job is slower than another. We measure the precision, relevance, and generality of the explanations generated.

Here are the two PXQL queries that we use:

1. **WhyLastTaskFaster:**
   `FOR` $T1, T2$
   `DESPITE` $jobID\_isSame = T \ \wedge \ inputsize\_compare = SIM \ \wedge \ hostname\_isSame = T$
   `OBSERVED` $duration\_compare = LT$
   `EXPECTED` $duration\_comare = SIM$
2. **WhySlowerDespiteSameNumInstances:**
   `FOR` $J1, J2$
   `DESPITE` $numinstances\_isSame = T \ \wedge \ pig\_script\_isSame = T$
   `OBSERVED` $duration\_compare = GT$
   `EXPECTED` $duration\_compare = SIM$

The first query asks why the last task on an instance runs faster than the earlier tasks that were executed on the same instance, even though each task processes a similar amount of data. Interestingly, we came across this query when we were puzzled by this pattern while collecting our experimental data. The reason we discovered was that the last task runs faster than earlier tasks because the instances have two cores and can run two tasks simultaneously. Sometimes, by the time the last task is reached, all other tasks are completed, and the instance is free to run only one task. Thus, the system load is lighter for the last task, and consequently the task is completed faster.

The second query asks why a job is slower than another job even though both jobs are running the same Pig script, and on the same number of instances. The explanation here is that the input size of the slower job is much greater than the input size of the faster job.

4.5.3 Well-specified PXQL Queries

In this section, we evaluate PerfXplain's explanation generation technique for PXQL queries where the user has specified a reasonable **des** clause as shown above. A good despite clause facilitates the generation of highly relevant explanations because the user manually constrains the search space. We compare the PerfXplain technique to the two naïve techniques described in Section 4.4.

Figure 4.3(a) shows the precision of the explanations generated by each technique for the **WhyLastTaskFaster** PXQL query. The x-axis indicates the width threshold specified for the explanation. Note that when the width is 0, the explanation is empty (or *true*) and thus the precision is $P(\mathbf{obs}|\mathbf{des} \wedge true) = P(\mathbf{obs}|\mathbf{des})$.

With the exception of a single run, both the RuleOfThumb approach and the PerfXplain approach generate the same explanation (for width 3), and thus achieve similar precision: `avg_cpu_user_isSame = F` $\wedge$ `avg_proc_total_isSame = F` $\wedge$ `avg_load_one_isSame = F`. The explanation says the task was faster because the average CPU time spent on user processes is not the same, the average total number of processes is not the same, and the average load time across a minute is not the same. This explanation is generally pointing towards the fact that the system load is different for the two tasks, and thus this leads to the observed behavior. (In the one exceptional run, PerfXplain generates an explanation starting with `avg_load_five_isSame = F`, which can also lead to faster execution, but it achieves a slightly lower precision than the above explanation.) The SimButDiff technique generates explanation `avg_pkts_in_isSame = F` $\wedge$ `avg_bytes_in_isSame = F` $\wedge$ `avg_pkts_out_isSame = F`. The first part says that the average number of packets arriving is different, the second part talks about average number of bytes in, and so on. This explanation is well-grounded in the data; it is indeed the case that if two tasks have a similar number of packets arriving, they also are likely to perform as expected (i.e. have a similar duration). However, not many pairs of tasks have a similar number of packets arriving, and it is not the case that just because two tasks have a different number of packets arriving that they will achieve very different runtimes. The predicate `avg_cpu_user_isSame = F` appears in SimButDiff's explanation, but usually not until the seventh or eighth predicate.

Query	**Avg Relevance Before**	**Avg Relevance After**
1	0.49	0.99
2	0.24	0.72

Table 4.3: Relevance of PXQL queries with an empty **despite** clause versus with a PerfXplain-generated **despite** clause.

Figure 4.3(b) shows the precision for the **WhySlowerDespiteSameNumInstances** task. For this task, the PerfXplain approach generates the following explanation (for width 3): `inputsize_compare = GT` $\wedge$ `avg_load_five_compare = GT` $\wedge$ `numinstances <= 12`. The first predicate in the explanation indicates that the input size is larger, which is the correct explanation. It continues to explain that the average load (measured at five-minute intervals) for the slower job is higher, which is probably just a result of the larger input size. Finally, it says that the number of instances is small. This is also correct because if both jobs had a sufficiently high number of instances, the change in data size would not have affected the runtime.

For the same task, the explanation of width 3 generated by the RuleOfThumb technique is `avg_load_five_isSame = F` $\wedge$ `avg_proc_total_isSame = F` $\wedge$ `inputsize_isSame = F`. The SimButDiff approach generates `inputsize_isSame = F` $\wedge$ `iosortfactor_isSame = T` $\wedge$ `blocksize_isSame = T`. Though both techniques note that the input size has an impact on the duration, they can only point to the fact that the input size is different (instead of greater than). Furthermore, the RuleOfThumb does not include `inputsize_isSame = F` until the third predicate because it is distracted by other side-effects of a larger input, which are that the average load is different and that the total number of processes is different.

In summary, we see that PerfXplain generates explanations with a better or equal average precision than the two naïve techniques. For example, for the **WhySlowerDespiteSameNumInstances** *query, at width* 3*, PerfXplain achieves at least 40.5% higher precision than both techniques.*

4.5.4 Under-specified PXQL Queries

In this section, we evaluate the quality of the **des** clauses generated by PerfXplain. We use the same PXQL queries as described in Section 4.5.2 but this time with the **des** clause removed. Table 4.3 above shows the relevance without the **des** clause, as well as the relevance with PerfXplain's automatically generated **des** clause. For this experiment, we restrict the clause to width 3.

Here are examples of **des** clauses that PerfXplain generates for each of the two queries:

1. `map_output_records_isSame = T` $\wedge$ `tracker_name_isSame = T` $\wedge$ `map_input_records_isSame = T` $\wedge$ `file_bytes_written_isSame = T`

2. `pigscript_isSame = T` $\wedge$ `numinstances_isSame = T` $\wedge$ `blocksize_isSame = T`

For the **WhyLastTaskFaster** query (1), we see that the **des** clause indicates that the numbers of map output records and the number of input records are the same for both tasks, as is the name of the tracker. The second and third predicates are analogous to the user-specified **des** clause. The user-specified **des** clause achieved an average relevance of 0.97, whereas the PerfXplain-generated one achieves 0.99. For the second query, we see that PerfXplain generates exactly the **des** clause that the user specified with the additional `blocksize_isSame = T` predicate at the end. Thus, it achieves a slightly higher relevance score of 0.72, compared to the user-specified **des** clause which has a relevance score of 0.6.

Figure 4.4(a) shows the relevance score for both PXQL queries for **des** widths ranging from 0 to 5. Once again, width 0 represents the relevance of the empty **des** clause. We see that for both PXQL queries, PerfXplain is able to generate **despite** clauses with high relevance.

In summary, we see that PerfXplain is able to generate a good **despite** *clause if the user fails to do so, thus increasing the relevance of an empty* **despite** *clause by up to 200%.*

4.5.5 Explaining a Different Job

In this section, we explore whether the techniques can support a scenario in which the pair of interest is different from all the jobs in the log. This experiment is trying to answer the question: Can we use the approach to explain the performance of new jobs, different from those executed in the past? We analyze this scenario for the **WhySlowerDespiteSameNumInstances** query. The pair of interest for this PXQL query consists of two jobs that are both running the same Pig script: simple-filter.pig. The log, however, consists only of the data for the simple-groupby.pig jobs (plus the pair of interest).

In this experiment, we execute the three explanation-generation algorithms over the log described above, and evaluate the explanation precision over a log consisting of all the simple-filter.pig jobs. Figure 4.3(c) presents the results.

Comparing the results to those in Figure 4.3(b), we see that PerfXplain performs slightly worse than when it has access to a normal job log. For width-1 explanations, the precision is significantly lower when PerfXplain has access to only simple-groupby.pig jobs (0.63) versus with the full log (0.93). However, by width-3, the difference shrinks to 0.02 (from 0.89 to 0.87). The average decrease in precision across the different widths for PerfXplain is 0.04. SimButDiff performs almost equivalently across the two scenarios, achieving a slightly lower precision (average of 0.001 lower). The average decrease in precision for the RuleOfThumb technique is 0.02.

In summary, we see that the precision of PerfXplain's explanations decrease slightly if the log consists of jobs that are different from the jobs in question. For example, for width-3 explanations we saw an average of only 2.7% decrease in precision.

4.5.6 Varying the Log Size

We investigate the effect of the log size on the different techniques. Namely, we randomly selected $x\%$ of the jobs in the log to use as the training log and varied x between 10% and 50%.

Figure 4.3(d) summarizes the results of the experiment for the **WhySlowerDespiteSameNumInstances** query. The x-axis represents the size of the log we used, and the y-axis reports the precision. The results shown are for width-3 explanations. We see that for the PerfXplain technique, the precision increases gradually with the size of the job log. However, even with just 10% of the log, PerfXplain achieves an average precision of 0.84.

However, the standard deviation of the precision at 10% is much higher than at 50% (0.08 versus 0.02). In contrast, the precisions of the RuleOfThumb and SimButDiff techniques are not significantly impacted by the sample size. In fact, the SimButDiff approach does not seem to be impacted by the sample size at all. The RuleOfThumb approach is affected by the size of the log, and the general trend is that it generates better explanations when the log is larger (with the exception of 0.4). However, the variance of its precision is high.

*A key takeaway from this experiment is that even with a small query log consisting of only 10% of the jobs, PerfXplain is able to achieve a high precision of 0.84 for width-*3 *explanations.*

4.5.7 Precision versus Generality Trade-off

As we discussed in Section 4.2.3, an effective explanation generation approach should achieve a good trade-off between precision and generality. Figure 4.4(b) plots the precision and generality scores of explanations generated by the different techniques. As the figure shows, PerfXplain achieves a better trade-off between generality and precision than the other approaches because its points fall higher and more to the right than the points for the other two approaches. (We connect PerfXplain's points to better show the positions of all its points in relation to the other techniques' points.)

4.5.8 Using Different Features

Using the right set of features is crucial to generating good explanations. On one hand, simpler features result in explanations that are generally applicable. On the other hand, having access to more complex features and just more features in general can lead to explanations that achieve higher precision.

In this section, we investigate the effect of different feature sets on the precision of the explanations that PerfXplain generates. We consider three different feature sets, we call them levels.

1. Level 1 includes only the `_isSame` features.

2. Level 2 includes the `_isSame` features, the `_compare` features, and the `_diff` features.

3. Level 3 includes the `_isSame` features, the `_compare` features, the `_diff` features, and the base features.

Figure 4.4(d) shows the precision of explanations for the **WhySlowerDespiteSameNumInstances** PXQL query at each feature level. As the figure shows, PerfXplain achieves a similar precision for both levels 2 and 3, which outperform level 1 by a significant margin. We see an improvement in precision for level 3 versus level 2, at width 3. Remember, that the explanation generated by PerfXplain for this approach is `inputsize_compare = GT` $\wedge$ `avg_load_five_compare = GT` $\wedge$ `numinstances <= 12`. The third predicate here says that it is because the number of instances is small. This feature is not included at Level 2, and thus we see the improvement at width 3.

In summary, we see that if we limit the set of features to only the `_isSame` *features, PerfXplain suffers significantly in terms of precision. However, hierarchy levels 2 and 3 perform similarly in this scenario.*

4.6 Conclusion

In this project, we addressed the problem of debugging performance problems in MapReduce computations. We presented PerfXplain, a system that enables users to ask comparative performance-related questions about either pairs of MapReduce jobs or pairs of MapReduce tasks. Our current implementation considers only queries over job or task runtimes but the approach can readily be applied to other performance metrics.

Given a performance query, PerfXplain uses a log of past MapReduce job executions to construct explanations in the form of predicates over job or task features. PerfXplain's key contributions include (1) a language for articulating performance-related queries, (2) a formal definition of a performance explanation together with three metrics, relevance, precision, and generality for explanation quality, and (3) an algorithm for generating high-quality explanations from the log of past executions.

Experiments on real MapReduce job executions on Amazon EC2 demonstrate that PerfXplain can indeed generate high-quality explanations, outperforming two naïve explanation-generation methods. While our focus in this chapter has been on MapReduce jobs, because PerfXplain simply represents job or task executions as feature vectors, the approach has the potential to generalize to other parallel data processing systems.

Algorithm 2 PerfXplain Algorithm.

Input: PXQL query $q = (\mathbf{des}, \mathbf{obs}, \mathbf{exp})$, jobs J_1 and J_2, set of all jobs J, width w

Output: an explanation X

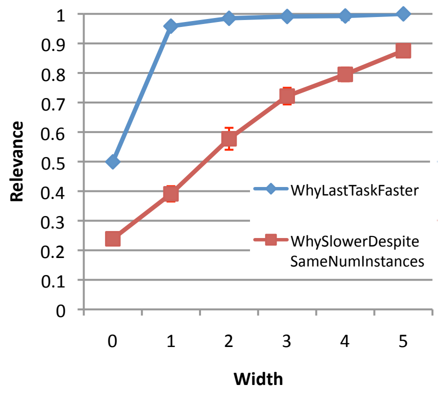
```
1: $P \leftarrow constructTrainingExamples(J, q)$
2: $P \leftarrow sample(P, J_1, J_2)$
3: $X \leftarrow true$
4: for $i = 1 \ldots w$ do
5:   $predicates \leftarrow [\ maxInfoGainPredicate(f)\ :\ f \in F\ ]$
6:   $precisions \leftarrow [\ P(obs|p, X)\ :\ p \in predicates]$
7:   $generalities \leftarrow [\ P(p|X)\ :\ p \in predicates\ ]$
8:   $predScores \leftarrow []$
9:   for $j \in [1 \ldots |predicates|]$ do
10:    $p \leftarrow predicates[j]$
11:    $precScore \leftarrow normalizeScore(precisions[j], precisions)$
12:    $genScore \leftarrow normalizeScore(generalities[j], generalities)$
13:    $predScores.append((p, w \cdot precScore + (1 - w) \cdot genScore)$
14:  end for
15:  $(bestPred, bestScore) \leftarrow \mathrm{argmax}_{p \in predScores}\, p.score$
16:  $X \leftarrow X \wedge bestPred$
17:  $P \leftarrow [\ p\ :\ p \in P \wedge X\ holds\ true\ for\ p]$
18: end for
19: return $X$
```

Algorithm 3 SimButDiff Algorithm.

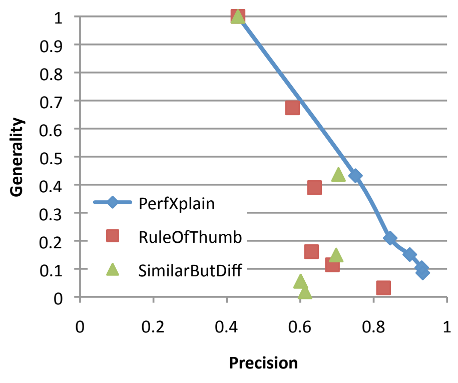
Input: PXQL query $q = (\mathbf{des}, \mathbf{obs}, \mathbf{exp})$, jobs J_1 and J_2, set of all jobs J, width w, similarity threshold s

Output: an explanation X

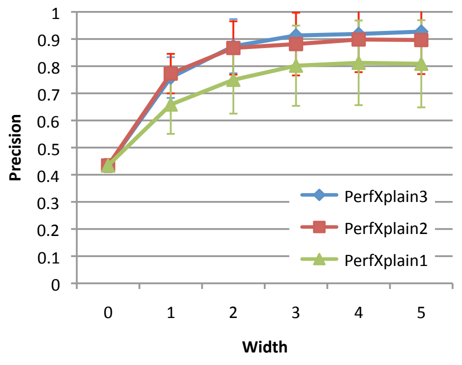
```
1:  $T \leftarrow constructTrainingExamples(J, q)$
2:  $isSameFeatures \leftarrow [\ f\ :\ f \in F \wedge f\ is\ a\ \_isSamefeature]$
3:  $T \leftarrow reduceDimensionality(T, isSameFeatures)$
4:  $k \leftarrow s \cdot dimensionality(T)$
5:  $S \leftarrow [\ t\ :\ t \in T \wedge t\ agrees\ with\ (J_1, J_2)\ on\ \geq\ k\ features]$
6:  $featureScores = []$
7:  for $f \in isSameFeatures$ do
8:      $d \leftarrow |[\ t\ :\ t \in S \wedge t.f \neq (J_1, J_2).f]|$
9:      $o \leftarrow |[\ t\ :\ t \in S \wedge t.f \neq (J_1, J_2).f \wedge \mathbf{exp}(t) = true]|$
10:     $featureScores.append((f, \frac{o}{d}))$
11: end for
12: $featureScores \leftarrow sort(featureScores)$
13: $X \leftarrow true$
14: for $i = 1 \ldots w$ do
15:     $(f, score) \leftarrow featureScores[i]$
16:     $X \leftarrow X \wedge (f = (J_1, J_2).f)$
17: end for
18: return $X$
```



(a)



(b)



(c)

Figure 4.4: (a) Relevance of PerfXplain-generated **despite** clauses, (b) Precision versus generality for **WhySlowerDespiteSameNumInstances**, (c) Precision of explanations for **WhySlowerDespiteSameNumInstances** with different feature levels.

Chapter 5

SIQ: GENERATING SAMPLE DATASETS FOR QUERY DEBUGGING

Successfully writing a correct query is an iterative process. With a question in mind, the user often begins with a simple `SELECT * FROM T` query to get a quick glance at the data. After the query finishes executing, the user looks at the data, examines the schema as well as the content, and returns to query writing. She might add a `WHERE` clause, join with another table, group on a column or two, and so on. So, in general, she modifies the query, executes the new query, waits for the query to finish executing, examines the output, and repeats this whole process until she has written her target query. Figure 5.1 summarizes this process. We refer to the sequence of queries that the user poses as a *query session*, or simply a *session*.

This process is often frustratingly slow, due to all three steps that occur in each iteration. First, modifying and writing SQL queries is difficult. Second, the query may take a long time to execute. Third, the queries are often over large datasets, which generally may lead to large output sizes. Thus, even examining the output can be an onerous task for the user.

A common solution employed by experienced users is to construct a sample dataset, sometimes called a toy database, which the user interacts with through the debugging process before executing their query over the full dataset. The toy database is constructed either by taking a small sample of the input data or by manually crafting it tuple-by-tuple. However, constructing a good toy database is a challenging endeavor. Naïvely selecting a random sample is insufficient, and leads to empty results while debugging, even for simple queries. As an example, consider the Internet Movie Database (IMDB). Suppose we have a small random sample of the Movies table (with schema `mid, title, year`), and the Genre table (with schema `mid, genre`). Even if the user writes a query simply joining the two tables, the output may be empty because the probability that there is a pair of tuples that

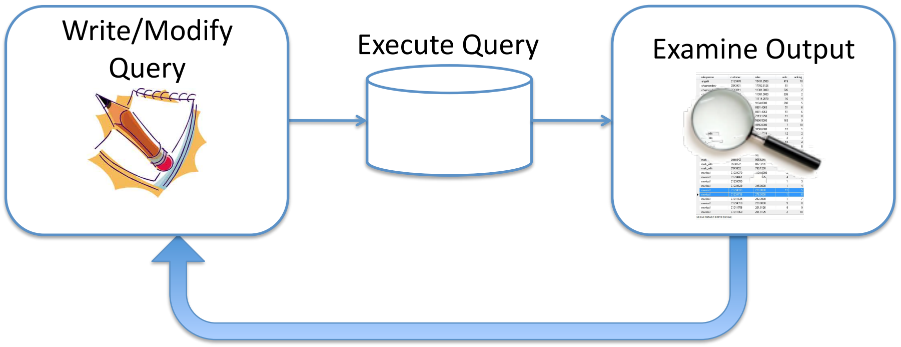


Figure 5.1: Iterative query debugging process. The sequence of queries that the user writes is referred to as a query session.

join in the two samples is low. A similar problem occurs with highly selective filters, such as a query that finds all actors whose last name is 'Theron'. Though manually constructing a toy database can solve this problem, it is a cumbersome task for the user. He or she must create each toy table, and then create each tuple by hand.

Prior work [107] by Olston *et al.* addresses this challenge with an algorithm for automatically generating a toy database for an input query (in the form of a Pig program). This approach, however, has a significant limitation. It is able to generate a toy database only once it has seen the query, and only for a single query. However, as described above, the query writing process is an iterative one. As such, if we apply Olston [108] directly, then each query in a session will have a distinct sample. This means that it will take the user some time in each iteration to examine the toy database and the query output. Therefore, it would be preferable to use the same toy database across all queries in the session. This way, the user would become familiar with the sample only once at the beginning of the session, and would then use it throughout the whole session. In this paper, we refer to this property as coherency (we define it formally in Section 5.1). As the user interacts with the toy database through each iteration of the query debugging process, we want the toy database to stay as coherent as possible (i.e., the dataset should change minimally across the queries in the session). We illustrate this point in Figure 5.2.

To summarize, a good toy database should ideally:

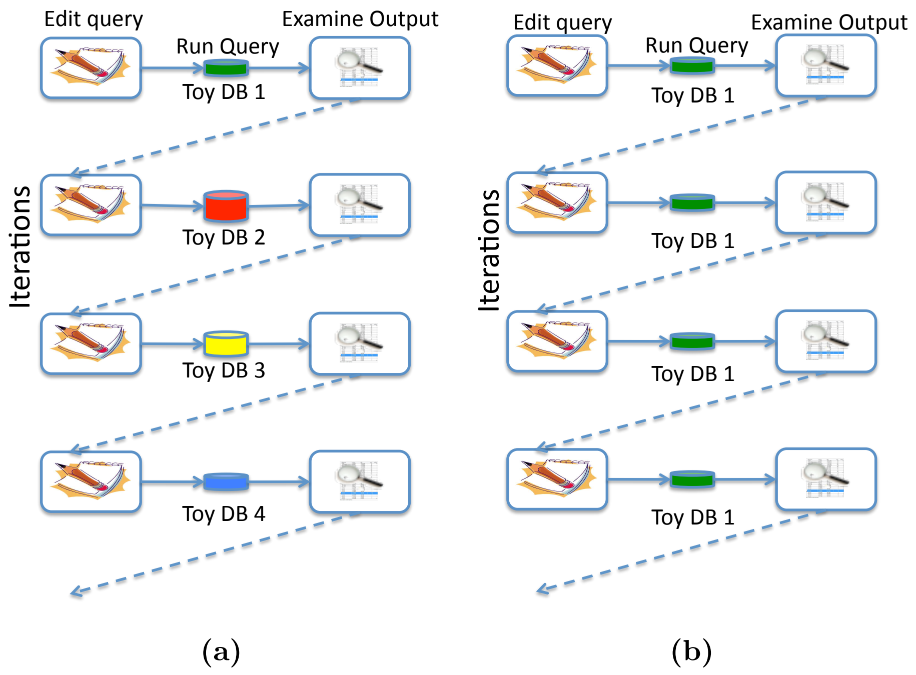


Figure 5.2: Directly applying Olston's algorithm generates a different sample for each query, and thus offers no coherency within a query session (a), whereas the ideal algorithm would allow the user to interact with the same sample through the session (b).

1. be small so that the user can familiarize herself with it quickly,
2. demonstrate the semantics of the operators in the query (e.g.., showing both tuples that pass a filter and tuples that do not pass the filter), so that the user can understand the query modifications they make, and quickly identify errors (we discuss this in more detail in Section 5.1), and
3. illustrate all the queries in a session so that the cost of familiarizing herself with the toy database is amortized across all queries in the session.

Generating a coherent toy database that illustrates all the queries in a session is more difficult than generating a toy database for a single query due to two additional challenges. First, we must generate the toy database prior to knowing the queries the user will articulate. Second, even if we did know exactly which queries the user is going to ask, we still face the challenge of generating a sample that is helpful for all the queries while still remaining small.

In this chapter, we present the Sample-based Interactive Query (SIQ) system. SIQ is able to generate a toy database, over which users can rapidly develop and debug their queries, before executing them over the full, original dataset. A small toy database enables users to more quickly formulate and debug their queries due to the instantaneous response times. Furthermore, the small size of the dataset means that users are able to more easily examine and verify the effects of their query edits. The toy database is constructed so that it illustrates the semantics of the user's query from the start (often a simple `SELECT *` query), and through all revisions of it (i.e., all queries in a query sesison). Additionally, the toy database remains coherent within a query session, so that the user's efforts can be directed toward query editing rather than monitoring the changes in the dataset. However, if needed, SIQ is able to repair a toy database, if the database is no longer illustrative of the user's query.

To summarize, the following is our problem statement:

Given a query session consisting of queries $q_1, \ldots, q_n$, generate a sequence of toy databases

$D_1, \ldots, D_n$ such that (1) D_i is illustrative of q_i, (2) D_i is generated prior to seeing D_{i+1}, (3) $D_1, \ldots, D_n$ are as small as possible, and (4) $D_1, \ldots, D_n$ are as coherent as possible.

The key insight behind the SIQ system is to use the log of all past queries executed over the full database to generate the smallest and most coherent toy databases for new queries. SIQ uses the log to try to guess where the user may be headed with their query, which helps increase the coherency and decrease the size of the toy database generated for each query.

Overall, we make the following contributions with SIQ:

1. We design and implement an algorithm, that utilizes a log of past queries, for generating a toy database with which the user can interact while debugging his or her query.

2. We define three quality metrics for what makes a good sequence of toy databases for a given query session: *completeness*, *conciseness*, and *coherency*. The first two are extensions of quality metrics from Olston *et al.* [107], whereas the third is new.

3. We design two techniques, *extending* and *regenerating*, for repairing a toy database which a user can invoke when the current toy database is no longer satisfactory for their needs.

4. We evaluate the SIQ system with real queries collected from an undergraduate database class. First, we find that SIQ is able to generate toy databases of similar quality to Olston's algorithm for single queries (Section 5.7.1). Second, we find that the toy database generated has varying levels of completeness as we vary how much of a query session SIQ knows a priori (Section 5.7.2). Third, we see that SIQ is able to offer coherent toy databases for query sessions while maintaining good completeness and conciseness (Section 5.7.3). Finally, we show that SIQ generates toy databases that are up to 85 times smaller than a naïve extension of Olston's algorithm (Section 5.7.4).

5.1 Definitions

In this section, we define a toy database, and articulate the properties that make a good sequence of toy databases for a given query session. We define these properties formally and illustrate the concepts with examples.

Definition 22 *A database D' is a toy database over a full database D, if D' and D share the same schema, D' follows the primary and foreign key constraints of D, and the tuples of D' are a subset of the tuples of D.*

Note that we restrict the toy databases that SIQ generates to the definition above. However, we relax the definition for when we compare against Olston's technique. The Olston algorithm sometimes generates fake tuples in its generated sample, and does not consider key constraints, and thus violates them occasionally.

Query sessions are defined as in Section 3.5. As an example, consider query 4 in Figure 5.3. In order to reach this query, the user begins with a simple `SELECT * FROM Actor` query, then adds the `gender` predicate to the `WHERE` clause, then adds the `Casts` table to the `FROM` clause along with the join predicate, and finally finishes with the `GROUP BY` clause. This sequence of queries executed by the user forms a single query session. The properties we define in this section are with respect to query sessions.

In this work, we assume that query sessions are append-only. Once a user has executed a query q_i as part of a session, any query q_j that follows it (i.e., $j > i$) will have only additional snippets of SQL added to it. For example, once the user adds a `GROUP BY` clause to the query, he or she will not remove it. This assumption is required for the following reason. When we generate a toy database, we aim to generate one that is illustrative of the target query that the user is going to write. However, since we do not know the target query a priori, we require a way to reduce the space of possible target queries. So, we assume that query sessions are append-only, and therefore can consider only those queries that have a superset of the current query's snippets.

We represent each SQL query as a relational algebra query plan, and the SIQ system works with these plans directly. To generate the query plan, we do not use the plan generated

1	SELECT	*
	FROM	Actor a
2	SELECT	*
	FROM	Actor a
	WHERE	*a.gender = ‘f’*
3	SELECT	*
	FROM	Actor a, *Casts C*
	WHERE	a.gender = ‘f’ *AND a.id = c.pid*
4	SELECT	*c.mid*, *count(*)*
	FROM	Actor a, Casts C
	WHERE	a.gender = ‘f’ AND a.id = c.pid
	GROUP BY	*c.mid*

Figure 5.3: Example of a query session for which we generate a toy database.

by the query optimizer, but rather a query plan that we refer to as the *canonical plan*. The canonical plan has *load base table* operators as leafs, above that it places joins first, then selections, then aggregates (if any), and finally projections. We currently do not support subqueries. SIQ presents to the user the toy database, the query plan, and how data is transformed by the plan into the output. Due to the canonical order of operators, the plans shown to the users have fewer variations than the plans generated by the optimizer. Figure 5.4 presents an example of such a query plan.

We now define our three quality metrics. Each metric is measured with respect to a sequence of toy databases and a query session. The first two metrics, completeness and conciseness, are borrowed from Olston [108], and are defined per query plan. To extend each metric to sessions, we simply measure its value for each query plan in the session and take the mean. We choose to pick the mean to stay consistent with how Olston [108] calculates the completeness and conciseness scores for a single query plan: by taking the mean of the scores at each node in the query plan. The third metric, coherency, is measured across all the query plans in a session.

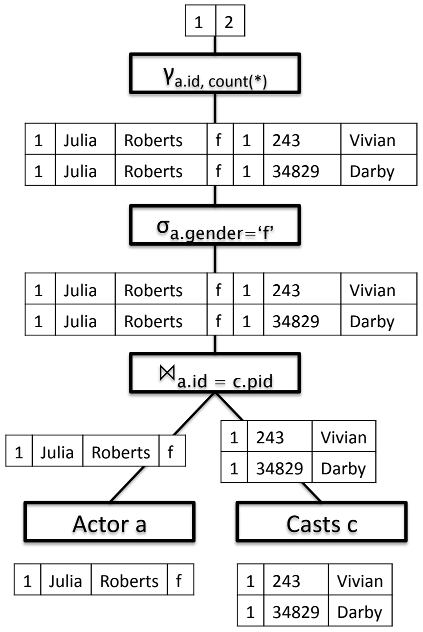


Figure 5.4: An example of a query plan with intermediate data.

5.1.1 Completeness

Completeness measures how well the sample data illustrates the various operators in the query plan. Each operator requires different types of tuples in order to be illustrated. For example, for a filter, a good sample is one which shows an example of a tuple that passes the filter and one that does not. However, a second tuple that passes the filter would be redundant. We say that the two tuples that pass the filter are in the same *equivalence class* [108].

Similar to Olston's work [108], for each operator, we define a set of equivalence classes as follows.

1. Load base table: every input tuple is assigned to a single equivalence class E_1. Our goal is to show at least one example tuple for the table.
2. Filter: every input tuple that passes the filter is assigned to an equivalence class E_1, and those that do not pass are assigned to equivalence class E_2. This ensures that we show one tuple that passes the filter, and one that does not.

3. Join: every output tuple is assigned to a single equivalence class E_1. The goal is to show at least one example of joining together tuples.
4. Project: every input tuple is assigned to a single equivalence class E_1, thus demonstrating the projection for at least one tuple.
5. Group by: every output record which has two or more underlying input records is assigned to equivalence class E_1. Our goal is to show an example of multiple input records being combined into one.

The equivalence classes for each operator is defined similarly to its corresponding Pig operator in Olston's work.

Definition 23 *Given an operator o and a toy database D, the* completeness score for the operator*, completeness(o, D) is the fraction of equivalence classes which contain at least one tuple.*

For example, in Figure 5.4, the join operator has completeness score 1.0, while the filter operator has only 0.5 because there is no tuple that fails the filter. Note that not all input or output records are assigned to an equivalence class. For example, for Group by, any output tuple that has just one underlying tuple is not assigned to any class.

5.1.2 Conciseness

Conciseness measures the fraction of tuples that are actually necessary to illustrate an operator's semantics. More precisely,

Definition 24 *Given an operator o and a toy database D, the* conciseness score for the operator*, conciseness(o, D) is the fraction of equivalences classes divided by the total number of example tuples at the operator.*

Maximizing conciseness indirectly leads to minimizing the size of the toy database. In Figure 5.4, the group-by operator has conciseness 1.0, while the join operator has a conciseness of 0.5 because there is only one equivalence class and two example tuples. Note how in this example, it is not possible to achieve a completeness score of 1.0 at the group-by operator while maintaining 1.0 conciseness at the join operator.

We extend these definitions to queries and query sessions. Completeness is extended in the following way. Conciseness is extended equivalently.

Definition 25 *Given a query q and a toy database D, the* completeness score for the query*, $completeness(q, D)$ is:*

$$completeness(q, D) = \frac{\sum_{i=0}^{k} completeness(o_i, D)}{n},$$

where $o_1, \ldots, o_k$ are the operators in q.

Definition 26 *Given a query session $s = q_1, \ldots, q_n$ and a sequence of toy databases $D_1, \ldots, D_n$, the* completeness score for the session*, $completeness(s, [D_1, \ldots, D_n])$ is:*

$$completeness(s, [D_1, \ldots, D_n]) = \frac{\sum_{i=0}^{n} completeness(q_i, D_i)}{n}.$$

5.1.3 Coherency

The coherency quality metric rewards sequences where the toy database stays consistent across iterations, throughout a query session. We define coherency as follows:

Definition 27 *Given a query session $s = q_1, \ldots, q_n$, and a sequence of toy databases $D_1, \ldots, D_n$. The* coherency *is:*

$$coherency(s, [D_1, \ldots, D_n]) = \frac{\sum_{i=1}^{n-1} Jaccard(D_i, D_{i+1})}{n-1},$$

where the Jaccard index of two sets D_i and D_{i+1} is $\frac{|D_i \cap D_{i+1}|}{|D_i \cup D_{i+1}|}$.

More informally, coherency is the mean set similarity between every adjacent pair of toy databases. The Jaccard index score is a popular metric used to measure the similarity between two sets.

5.2 Naïve Approach

Our goal is to construct a toy database that is illustrative of all the queries in a user's current session after seeing only the first query in that session. We demonstrated in the earlier chapters that past queries can help users articulate future queries. We propose to leverage this observation for the purpose of query illustration by building a toy database that is illustrative of previous sessions, with the hypothesis that the toy database will also be illustrative of future query sessions. We show in Section 5.7.3 that SIQ is indeed able to generate toy databases, from a query log, that achieves good completeness and conciseness for a random set of ten queries, which are not from the log.

As such, we reduce our problem to the following. *Given a set of past query sessions, and thus a set of queries $Q_1, \ldots, Q_n$ over a database D, construct a toy database that achieves a high completeness and conciseness score across these queries.*

These past query sessions can be either (1) all past query sessions in the log or (2) the subset of past query sessions that are possible extensions to the query that the user has typed so far. In Chapter 3, we showed how to effectively extract the latter subset of query sessions. In this chapter, we assume the subset if given.

An initial approach is to convert each query into a Pig Latin dataflow program, to use Olston's technique[108] to illustrate each program, and then to take the union of all these datasets. Though we do not try this technique directly, we implement a variation of our algorithm where we use our technique on a per-query basis and then take the union of all the samples. We compare our technique on a per query basis to the Olston technique in Section 5.7.1, and find that the two techniques generate samples of similar quality for single queries. After this comparison, we then show in Section 5.7.4 that this union-based technique results in toy databases that are orders of magnitudes larger than the toy databases that SIQ generates.

Consider the following simple example. Suppose that in order to illustrate q_1, we require a single female `actor` tuple and thus we select `actor(546420, Drew, Barrymore, f)`. Suppose for q_2 we need an actor with last name 'Roberts' and thus we have selected `actor(401326, Arnold, Roberts, m)`. Taking the union, results in a toy database consisting of two tuples. However, if we had been more careful, we could have satisfied both requirements with a single tuple where the last name is 'Roberts' and the gender is 'f', such as `actor(770247, Julia, Roberts, f)`.

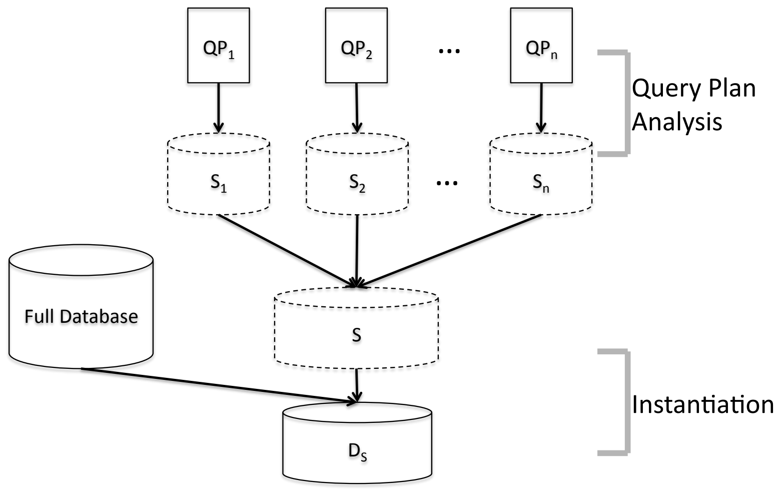


Figure 5.5: SIQ algorithm overview. Each QP_i is a query plan, each S_i is a symbolic database, and D_S is the generated toy database.

5.3 *SIQ Approach*

At a high level, the SIQ approach consists of two phases. In the first phase, the *Query Plan Analysis* phase, we generate a symbolic database per query that describes the kind of tuples required to illustrate each query. Informally, a symbolic database is like a database except we have variables instead of values. We then merge the symbolic databases into one large symbolic database. In the second phase, *Instantiation*, we find an instantiation of the symbolic database consisting of real tuples from the database. Figure 5.5 summarizes this process.

Before we continue, we introduce a few definitions. Note that symbolic databases are defined similarly to incomplete databases in Imielinski and Lipski [81], as well as symbolic databases in Binnig *et al.* [30]. Symbolic databases are most similar to v-tables as defined by Imielinski and Lipski, but also allow constraints on variables (that only refer to a single variable), as well as constraints specifying that a pair of tuples may not be merged together.

Definition 28 *A* symbolic tuple *is a k-tuple $(v_1, \ldots, v_k)$ where each v_i is a variable. Associated with each variable v is a set of constraints that we refer to as $constraints_v$. We restrict $constraints_v$ to be a conjunction of atomic predicates of the form: v op c where c is a constraint and op is $=, \neq, <, >, \leq,$ or $\geq$.*

Definition 29 *A* symbolic relation *is a relation schema (defined traditionally) together with a set of symbolic tuples.*

Definition 30 *A* symbolic database *S is a set of symbolic relations $R_1, \ldots, R_n$ together with a `NoMerge` set, which represents the pairs of tuples that may not be instantiated with the same tuple. `NoMerge` is a set of pairs (t_1, t_2) where t_1 and t_2 are tuples that appear in some symbolic relation R_i, $1 \leq i \leq n$.*

We refer to all the tuples in all the relations of a symbolic database S as $tuples(S)$.

We now define what it means for a database instance to *instantiate* a symbolic database.

Definition 31 *A database instance D* instantiates *a symbolic database S if there is a mapping $m : tuples(S) \rightarrow tuples(D)$ such that:*

1. *if $m(t) = t'$, then t and t' have the same schema,*
2. *if $m((v_1, \ldots, v_n)) = (c_1, \ldots, c_n)$, then $constraints_{v_i}(c_i) = true$ for $1 \leq i \leq n$, and*
3. *if $m(t_1) = t$ and $m(t_2) = t$, then $(t_1, t_2) \notin$ `NoMerge`.*

If the mapping is partial (i.e., it is undefined for some tuples in S), then we say that D is a partial instantiation *of S.*

Note that there exist symbolic databases that can not be instantiated. For example, if there is a symbolic tuple that says that the value of column c is v, and if there is no such tuple in the full database.

As an example consider the symbolic database presented in Figure 5.6, but suppose it only consists of the three `Movie` tuples m_1, m_2, and m_3. If we disregard the `NoMerge`

Actor

	id	fname	lname	gender
a_1	x_1	x_2	x_3	x_4 = ‘f’
a_2	x_5	x_6	x_7 = ‘Roberts’	x_8

NoMerge:
<m_1, m_2>

Casts

	pid	mid	role
c_1	x_1	x_9	x_{10}
c_2	x_1	x_{11}	x_{12}

Movie

	id	name	year
m_1	x_9	x_{13}	x_{14}
m_2	x_{11}	x_{15}	x_{16}
m_3	x_{17}	x_{18}	$x_{19} > 1990$

Figure 5.6: An example of a symbolic database.

Movie

	id	name	year
t_1	9458	Citizen Kane	1941
t_2	2473	The Shawshank Redemption	1994

Figure 5.7: An incorrect instantiation of m_1, m_2, m_3 from Figure 5.6. This is because $(m_1, m_2) \in$ `NoMerge`.

component, then the database shown in Figure 5.7 is a correct instantiation; we map m_1 and m_2 to t_1, and m_3 to t_2. However, if we consider the `NoMerge`, then this is not a correct instantiation because $(m_1, m_2) \in$ `NoMerge`, and thus they can not be mapped to the same tuple.

5.4 *Query Plan Analysis*

In this section, we describe the Query Plan Analysis (QPA) phase of the SIQ approach. The algorithm we present takes a query plan Q as input and produces a symbolic database S. The symbolic database S is such that if we can instantiate it fully with a database D, then D should have perfect completeness with respect to Q.

Algorithm 4 outlines the QPA phase. The algorithm begins by constructing an empty symbolic database (line 1), and this database gets populated as the algorithm proceeds.

Algorithm 4 Query Plan Analysis (QPA) algorithm.

```
Input: query plan $Q$
Output: a symbolic database $S$
1: $S \leftarrow$ empty symbolic database
2: $toBeAnalyzed \leftarrow [Q.root]$
3: while $toBeAnalyzed \neq \emptyset$ do
4:     $op \leftarrow toBeAnalyzed.dequeue()$
5:     for $c \in equivalenceClasses(op)$ do
6:         $request(op, canonicalTuple(c), S)$
7:     end for
8:     for $op' \in childrenOps(op)$ do
9:         $toBeAnalyzed.enqueue(op')$
10:    end for
11: end while
12: return $D_S$
```

The algorithm begins by processing the root operator in the query plan (line 2). It first identifies the equivalence classes for the operator, and requests one tuple per equivalence class (lines 5-7). For example, for the filter operator in Figure 5.4, it requests the tuple $(x_1, x_2, x_3, x_4 =$'f'$, x_5, x_6, x_7)$ and the tuple $(y_1, y_2, y_3, y_4 \neq$'f'$, y_5, y_6, y_7)$. When a tuple t is requested from an operator *op*, the requested tuples are translated down through the query plan until we know which tuples to add to the base tables in order for t to appear as part of *op*'s intermediate data. We discuss its details shortly. Once we have added the tuples for this operator, we then look at its children operator(s) and add them to our *toBeAnalyzed* queue (lines 8-10). We repeat this process. So, in the second iteration we process the children operator(s) of the root operator, and then the children operator(s) of the children operator(s), and so on. We continue until *toBeAnalyzed* is empty, which means that we have processed all the operators in our query plan.

The *request* method takes an operator *op*, a requested tuple $t = (x_1, \ldots, x_n)$ along with its associated constraints, and a symbolic database S as input. Let's suppose that t has schema $(c_1, \ldots, c_n)$. It finishes when it has added a few tuples to S that cause t to appear in the intermediate data at *op*.

1. If *op* is a load table operator for a relation R, then add t to the R relation in S.

2. If *op* is a filter operator with predicate c_i op' v for some value v, then request tuple $(x_1, \ldots, x_i \ op' \ v, x_{i+1}, x_n)$ from the child operator.
3. If *op* is a group by operator, SIQ currently only supports the `count` aggregate, so then it must be the case that the group by is on columns $c_1, \ldots, c_{n-1}$ and that x_n is an integer (i.e., the desired count). Generate x_n tuples $t_1, \ldots, t_{x_n}$ conforming to the child operator's schema, and copying over the variables from the columns $c_1, \ldots, c_{n-1}$. Fill the remaining columns with new variables (with no constraints). Call request on the child operator for each t_i. Afterward, add every pair (t_i, t_j) to the `NoMerge` list for S.
4. If *op* is a cartesian product operator, and the schemas of the children operators op_1 and op_2 are $c_1, \ldots, c_j$ and $c_{j+1}, \ldots, c_n$, respectively, then request $(x_1, \ldots, x_j)$ from op_1, and $(x_{j+1}, \ldots, x_n)$ from op_2.
5. If *op* is a join operator, the schemas of the children operators op_1 and op_2 are $c_1, \ldots, c_k$ and $c_{k+1}, \ldots, c_n$ respectively, and the join predicate is $c_i = c_j$, then request $(x_1, \ldots, x_i, \ldots, x_k)$ from op_1, and $(x_{k+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n)$ from op_2. Note how we have replaced the reference to variable x_j with a reference to x_i instead.
6. If *op* is a project operator and the underlying schema is $c_1, \ldots, c_n, c_{n+1}, \ldots, c_m$, then request $(x_1, \ldots, x_n, y_1, \ldots, y_{m-n})$ from the child operator, where each y_i is a new variable (for $1 \leq i \leq m - n$).

5.5 Instantiation

Given a symbolic database S and the full database instance, the goal of the Instantiation phase is to find a partial instantiation D of S that includes as many tuples of S as possible in its mapping and has few tuples. We aim to instantiate many tuples of S with the goal of achieving high completeness in future sessions, and for few tuples in D as this often leads to higher conciseness.

We begin by describing an algorithm that finds a partial instantiation without concern for the size of D. We then extend this to a greedy algorithm that is able to find a small instantiation.

```
SELECT   *
FROM     Actor a1, Actor a2,
         Casts c1, Casts c2,
         Movie m1, Movie m2, Movie m3
WHERE    a1.gender='f' AND a2.lname='Roberts'
         AND m3.year > 1990
         AND a1.id = c1.pid AND a1.id = c2.pid
         AND c1.mid = m1.id
         AND c2.mid = m2.id
LIMIT    1
```

Figure 5.8: SQL query translation of Figure 5.6.

5.5.1 Finding Any Instantiation

We begin by describing a direct translation technique that takes a single symbolic database S and translates it into a single SQL query. The output of this SQL query is the set of all instantiations of S over the underlying database. We follow up with a technique which generates multiple SQL queries, which is both faster and is able to find partial instantiations for symbolic databases that can not be fully instantiated.

To demonstrate the translation from symbolic database to SQL query, we show the translation for an example symbolic database. Consider the symbolic database S in Figure 5.6. The translation works as follows. For each symbolic tuple in S, we add a table to the `FROM` clause. From there, every constraint for a variable is added to the query as a predicate in the `WHERE` clause. For example, the constraint $x_4 = `f'$ is translated to the predicate `a1.gender = 'f'`. Next, multiple occurrences of a variable are also translated into predicates. More specifically, if a variable appears in two cells, then we add a predicate to enforce that these two columns are equal. For example, we can see in Figure 5.6 that $x_1$ appears in $a_1$'s `id` column but also in $c_1$ and $c_2$'s `pid` column. This is translated into the equality predicates `a1.id = c1.pid` and `a1.id = c2.pid`. Finally, we select all the columns

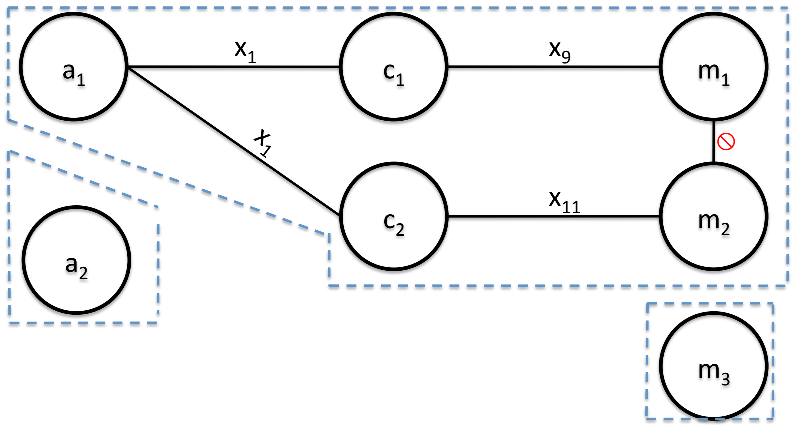


Figure 5.9: Graph representing the symbolic database in Figure 5.6.

and add the `LIMIT` clause because we only need one instantiation of the symbolic database. We refer to this algorithm as $findFullInstantiation$ in the remainder of this section.

Optimization: The problem with generating a single SQL query is that the number of joins in the query grows linearly with the number of symbolic tuples. Thus, a symbolic database with n tuples results in a SQL query with an n-way join. Furthermore, if there is any part of the symbolic database that can not be instantiated, then the query returns no result. However, there is no reason that we should limit ourselves to generating a single SQL query.

In this section, we describe a better algorithm for instantiating a symbolic database S that executes multiple SQL queries, which we call simply the $findInstantiations$ algorithm. The technique is outlined in pseudocode in Algorithm 5. The method takes as input the symbolic database S and outputs a database D_S that instantiates S. First, it represents S as a graph G (lines 1-3). There is a vertex for every tuple (line 1) and there is an edge between two tuples if either they share a variable or if they can not be merged (line 2). Figure 5.9 shows this graph representation for the symbolic database found in Figure 5.6. We then partition G into its connected components (line 4). Each connected component defines a smaller symbolic database consisting only of the tuples in the component (line 5). We then execute the $findFullInstantiation$ algorithm, as described above, on each

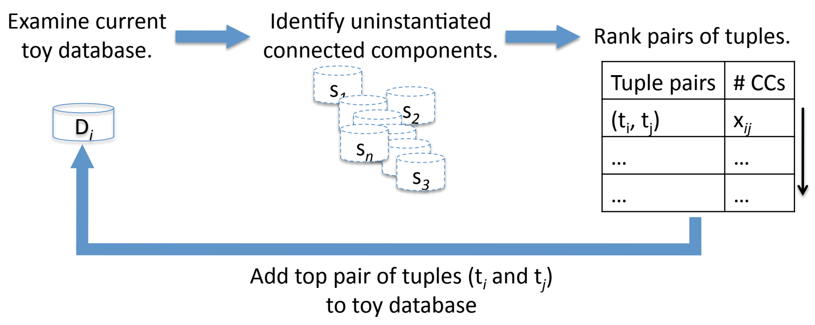

Examine current toy database.
Identify uninstantiated connected components.
Rank pairs of tuples.
D_i
S_1
S_2
S_n
S_3
Tuple pairs
# CCs
(t_i, t_j)
x_ij
...
Add top pair of tuples (t_i and t_j) to toy database

Figure 5.10: Overview of technique for building a small toy database.

connected component and take the union of these instantiations (lines 6-9). Finally, we return the resulting database D_S (line 10).

Algorithm 5 $findInstantiations$ algorithm breaks symbolic database into connected components.

```
Input: symbolic database $S$
Output: a database $D_S$ that instantiates $S$
 1: $V \leftarrow \{t : t \text{ is a symbolic tuple in } S\}$
 2: $E \leftarrow \{(u,v) \in V^2 : (vars(u) \cap vars(v) \neq \emptyset) \vee noMerge(u,v)\}$
 3: $G \leftarrow (V,E)$
 4: $\{G_1,\ldots,G_n\} \leftarrow connectedComponents(G)$
 5: $\{S_1,\ldots,S_n\} \leftarrow \{S_i : \text{symbolic database consisting of tuples } G_i.V\}$
 6: $D_S \leftarrow \emptyset$
 7: for $i \in [1\ldots n]$ do
 8:     $D_S \leftarrow D_S \cup findFullInstantiation(S_i)$
 9: end for
10: return $D_S$
```

5.5.2 Finding a Small Instantiation

In this section, we describe how to find a small instantiation of a symbolic database. The technique utilizes the algorithms from the above section.

At a high-level, we build the instantiation iteratively, using a greedy algorithm. Given the symbolic database S, we first break it into its connected components $s_1,\ldots,s_n$. Then we begin with an empty toy database D_0. In each iteration, we do the following. For every pair of tuples t_i, t_j from the full database D (that are not already in the toy database), we count the number of connected components that would be successfully instantiated if we were to add the tuples to the toy database. It is not computationally feasible to compute this for every pair, so we use sampling, which we describe in the Optimization part at the end of this section. Then we pick the pair of tuples with the highest such score to add. We repeat the process until the number of instantiated connected components does not increase. The reason we consider pairs of tuples, instead of one tuple at a time, is to reduce the chance of terminating early due to a local minimum. Informally, considering two tuples at a time makes the algorithm less greedy. Figure 5.10 summarizes this process.

We now describe the algorithm in more detail. Algorithm 6 outlines the pseudocode.

Algorithm 6 $findSmallInstantiation$ algorithm.

```
Input: symbolic database $S$
Output: a database $D_S$ that instantiates $S$
 1: $D_S \leftarrow \emptyset$
 2: $\{G_1, \ldots, G_n\} \leftarrow connectedComponents(G)$
 3: $U \leftarrow \{S_i$ : symbolic database consisting of tuples $G_i.V\}$
 4: while $U \neq \emptyset$ do
 5:     $t_1, t_2 \leftarrow findPairThatInstantiatesMostCCs(U, D_S)$
 6:     $score \leftarrow |\{S_i \in U : findFullInstantiation(S_i, D_S \cup \{t_1, t_2\}) \neq \emptyset\}|$
 7:     $score1 \leftarrow |\{S_i \in U : findFullInstantiation(S_i, D_S \cup \{t_1\}) \neq \emptyset\}|$
 8:     if $score > 0$ then
 9:         $D_S \leftarrow D_S \cup \{t_1\}$
10:         if $score > score1$ then
11:             $D_S \leftarrow D_S \cup \{t_2\}$
12:         end if
13:     end if
14:     $instantiated \leftarrow \{S_i \in U : findFullInstantiation(S_i, D_S) \neq \emptyset\}$
15:     $U' \leftarrow U \setminus instantiated$
16:     if $|U'| = |U|$ then
17:         terminate
18:     end if
19: end while
20: return $D_S$
```

The $findSmallInstantiation$ algorithm starts with the empty toy database D_S (line 1). Next, it identifies the connected components of the symbolic database (lines 1-3). Henceforth, we refer to the connected components as *cc*'s, for short. We maintain the list of uninstantiated *cc*'s as U. Next, while there exists some uninstantiated *cc*, we repeatedly add tuples to our sample (the *while* loop from lines 4-19). Each iteration of this *while* loop begins by identifying the pair of tuples that, if added to the sample D_S, would instantiate the largest number of *cc*'s. In other words, we are finding:

$$(t_1, t_2) \leftarrow \underset{t_i, t_j}{\operatorname{argmax}} |\{S_i \in S : (D_S \cup \{t_1, t_2\}) \ instantiates \ S_i\}|.$$

This $findPairThatInstantiatesMostCCs$ step (line 5) is implemented using SQL queries, as follows. First, for every pair t_1, t_2, we count the number of *cc*'s that they can instantiate together with the help of the existing sample D_S. We call this the *teamScore*. We compute the *teamScore* as follows. For a given *cc*, if we take the SQL query that finds its instantiations (as in Figure 5.8), with a few modifications, we can find all pairs of tuples, that if added to the sample, would satisfy the *cc*. We do this translation for every *cc*, take the union of all the results, and group by the tuple identifier columns, the `oid`'s, of the pair of tuples. This results in a long and slow SQL query however. We describe two optimization techniques at the end of this section.

The second score we compute is the *individualScore*, which is the number of *cc*'s that either t_1 or t_2 is able to instantiate with the existing sample. We use similar technique as above to find the *cc*'s that can be instantiated by adding a single tuple. Then for each pair of tuples, we take the union of their respective *cc*'s, and the size of their union gives us their *individualScore*. We then add the *teamScore* and the *individualScore* to give us the final score for each pair. We take the pair with the highest score (line 6).

If the score is greater than 0 (i.e., at least some additional *cc*'s are instantiated), then we add t_1 to the sample (lines 8,9). It is sometimes the case that the score for a pair of tuples t_1 and t_2 is exactly the same as the score of adding a single tuple t_1. For example, if t_1 alone instantiates k *cc*'s and t_2 instantiates a subset of those *cc*'s, then the combined score of the pair is k. In such cases, we simply add t_1 to the sample and discard t_2 for this iteration (lines 10-12).

At the end of this iteration, we identify all the *cc*'s that are now instantiated (line 14), compute the set difference with U (line 15), and call this U'. If U' is the same size as U

(i.e., no additional *cc*'s were instantiated in this round), then we stop the algorithm (lines 16,17). Otherwise, we continue and repeat this process.

Optimization: We describe three techniques for optimizing the query that finds the *teamScore* for every pair of tuples. First, many *cc*'s are repeated across the query log, so instead of repeating every occurrence in our SQL query, we simply incorporate the number of repetitions into our query. Thus, when we group by the pair of `oid`'s, instead of counting (the number of *cc*'s), we take the sum of the numbers of repetitions. Second, not every *cc* needs to be included in this computation in every iteration. Therefore, in a given iteration, we include in the SQL query only the most popular *cc*'s until we have covered at least k% of the total number of *cc*'s. Our current implementation has k set at 50%, but this threshold is easily adjustable. Finally, our third optimization is to, for every *cc*, include only $\frac{1}{x}$ of the pairs of tuples, that if added would satisfy the *cc*. We do this by adding a predicate to the SQL query that says `(oid1 + oid2)%x = 0`. Though this will fail to find some optimal pairs of tuples, it is better than doing a simple `LIMIT` on the query because if a pair of tuples satisfies two *cc*'s and it appears in the list for one of the *cc*'s, then it is guaranteed to appear in the list for the second *cc* as well. Thus, informally, we are using the limited spots more efficiently.

5.6 Repairing the toy database

Despite the quality of the toy database, it will sometimes be insufficient at illustrating the user's query session. As such, repairing a toy database is of high importance. In this work, we support two techniques for repairing a toy database: *regenerating* and *extending*. The repair process is currently user-invoked.

We begin by defining the notion of a *potential target query* for a query q. We say that a query q' is a potential target query if q' appears in the query log, and can be written by starting with q and only adding snippets of SQL. This is where our assumption, which says that sessions are append-only, is useful.

Regenerating a toy database involves discarding the current sample and generating a new toy database. For this approach, we can reuse the existing algorithm as described in Algorithm 6. However, we now have additional information from the user: the query session thus far. Therefore, when generating the new sample, we no longer consider the full query log. We only consider queries that are potential target queries of the user's current query.

Extending a toy database consists of starting with the existing current sample, and extending it until we have a high-quality toy database again. Though this technique may lead to lower conciseness, it will result in better coherency across iterations. For extending, we can again reuse the algorithm described in Algorithm 6. However, instead of starting with the empty sample (line 1), we begin with the existing sample.

5.7 Evaluation

To evaluate SIQ, we run four different experiments, using the IMDB database along with a query log over it. We used this dataset for evaluating SnipSuggest, and thus we described it earlier in Section 3.6.

First, we compare the SIQ algorithm and the Olston technique for single queries. We examine the sample datasets generated by the two algorithms for six different queries of varying complexity. We show that the SIQ algorithm performs comparably to the Olston algorithm for single queries.

In our second experiment, we study how knowing the future can assist in generating toy databases. For a given query session, we vary how much of the query session we know a priori, to see the impact that this knowledge has on the completion and conciseness of the dataset.

In our third experiment, we compare SIQ to using the Olston algorithm at every step of a query session. We do this for ten randomly selected query sessions, and we examine the completeness, conciseness, and coherency of the sequence of toy databases.

Finally, we compare the SIQ technique to the union-based technique that we described in Section 5.2. We measure how the size of the toy database changes as the workload sizes increases.

5.7.1 Single-Query Case Study

In this section, we consider six different queries of varying complexity, and perform a case study of the two techniques. We use the SIQ technique to generate a toy database given

Query	Toy Database Generated

`SELECT *`
`FROM Actor a`
	Actor:
	id | **fname** | **lname** | **gender**
	3244 | rodney | afif | m
	Com: 1.0 Con: 1.0 Real: 1.0

`a= LOAD ‘Actor.csv’ USING PigStorage(‘,’)`
`AS (id, fname, lname, gender);`
`ILLUSTRATE a;`
	Actor:
	id | **fname** | **lname** | **gender**
	5080 | Mohamed | Al-Fayed | m
	Com: 1.0 Con: 1.0 Real: 1.0

Figure 5.11: The samples generated by the SIQ technique versus the Olston technique for a simple `SELECT *` query [**Query 1**].

a single query, and we do the same with the Olston approach. We measure conciseness, completeness, and realism (the fraction of tuples in the toy database that appear in the full IMDB database). Note that the definition of a toy database allows only databases with 100% realism. However, because Olston's technique sometimes generates fake data, we allow fake data and measure realism in this experiment.

Query 1: We begin with a simple `SELECT *` query. Figure 5.11 shows the query, its translation into Pig, the samples generated by the two techniques, and the quality scores for the samples. We see that both techniques generate just one tuple to illustrate this query, thus achievning a score of 1.0 for completeness, conciseness, and realism (Com, Con and Real in the figure, respectively). We truncate certain column values for cleaner presentation.

Query 2: Next, we examine a query over the `Movie` table with a simple filter on the movie name in Figure 5.12. Even with such a simple query, we notice a few interesting points. First, the sample generated by the SIQ algorithm has perfect completeness, but has a conciseness of 0.75. The reason is that there are two operators in the query. The first is the `LOAD` operator, which requires only a single tuple to illustrate its semantics and thus the sample has a conciseness of 0.5 for this operator. Whereas for the `FILTER`, we need both tuples and the conciseness is 1.0. Thus, the total conciseness is $(0.5 + 1.0)/2 = 0.75$, which is in fact the highest conciseness we can have while achieving perfect completeness.

Query | **Toy Database Generated**

```
SELECT   *
FROM     Movie m
WHERE    name = ‘Cold Mountain’
```

Movie:

id	**name**	**year**
66947	Cold Mountain	2003
289197	Sawing Wood	1896

Com: 1.0 Con: 0.75 Real: 1.0

```
m= LOAD ‘Movie.csv’ USING PigStorage(‘,’)
        AS (id, name, year);
f = FILTER m BY name == ‘Cold Mountain’;
ILLUSTRATE f;
```

Movie:

id	**name**	**year**
239692	Old Maid’s First	1903
239692	Cold Mountain	1903

Com: 1.0 Con: 0.75 Real: 0.5

Figure 5.12: The samples generated by the SIQ technique versus the Olston technique for a simple query with a single filter [**Query 2**].

The second interesting point is that the Olston algorithm generates fake data to illustrate this query. This happens because when the algorithm is generating a toy database, it begins with an initial sample of the original dataset. Any toy database generated by Olston’s algorithm consists of tuples that either appear in this initial sample or are fake. Therefore, since there is no ‘Cold Mountain’ tuple in this initial sample, the algorithm constructs such a tuple itself.

The third interesting pattern follows from the second. While generating the fake tuple, the Olston algorithm has violated a primary key constraint (i.e., there are two distinct tuples with the same `id` in the Movie table).

Query 3: In Figure 5.13, we examine a query with two predicates in the `WHERE` clause. Note that we now need three tuples. One that fails both predicates, one that satisfies the ‘Julia’ predicate but fails the ‘Roberts’ predicate, and one that satisfies both predicates.

The SIQ algorithm generates such tuples, thus achieving a completeness score of 1.0. The conciseness is 0.66, which, like Query 2, is the maximum conciseness we can achieve while maintaining a completeness of 1.0.

The Olston algorithm also generates a sample with completeness 1.0 and conciseness

0.66, but again with the help of fake data (which again leads to a primary key violation).

Query

```
SELECT   *
FROM     Actor a
WHERE    fname = ‘Julia’ AND
         lname = ‘Roberts’
```

Toy Database Generated

Actor:

id	fname	lname	gender
639883	Julia	Griffin	f
4936	E.	Akopov	m
770247	Julia	Roberts	f

Com: 1.0 Con: 0.66 Real: 1.0

Query

```
a= LOAD ‘Actor.csv’ USING PigStorage(‘,’)
          AS (id, fname, lname, gender);
f1 = FILTER a BY fname == ‘Julia’;
f2 = FILTER f1 BY lname == ‘Roberts’;
ILLUSTRATE f2;
```

Toy Database Generated

Actor:

id	fname	lname	gender
3542	Vatentin	Agopov	m
3542	Julia	Roberts	m
3542	Julia	0	m

Com: 1.0 Con: 0.66 Real: 0.33

Figure 5.13: The samples generated by the SIQ technique versus the Olston technique for a simple query with two filters [**Query 3**].

Query 4: For the fourth query, we include a join. Figure 5.14 shows the query and the samples. The SIQ algorithms sample is simply two tuples that can be joined together. The Olston algorithm is similar, but somehow includes both tuples twice (our belief is that this is due to a bug).

Query 5: We now also include a `GROUP BY` clause. The SIQ algorithm generates a sample with perfect completeness, and the maximal possible conciseness in this case. If we improved the Olston technique to remove duplicates, it would achieve a conciseness score of 0.88 (higher than the SIQ algorithm's 0.625). However, this would come at the cost of a foreign-key violation.

Query 6: Our final query includes a three-way join, grouping, and an additional filter (Figure 5.16). Once again, we see that SIQ's sample achieves a 1.0 completeness and a lower conciseness (that is still the highest possible conciseness that we can achieve).

Query | **Toy Database Generated**

```
SELECT   *
FROM     Actor a, Casts c
WHERE    a.id = c.pid
```

Actor:

id	**fname**	**lname**	**gender**
3244	Rodney	Afif	m

Casts:

pid	**mid**	**role**
3244	137560	Marty Vella

Com: 1.0 Con: 1.0 Real: 1.0

```
a= LOAD ‘Actor.csv’ USING PigStorage(‘,’)
        AS (id, fname, lname, gender);
c= LOAD ‘Casts.csv’ USING PigStorage(‘,’)
        AS (pid, mid, role);
j = JOIN a BY id, c BY pid;
ILLUSTRATE j;
```

Actor:

id	**fname**	**lname**	**gender**
938	Thomas	Abernathy	m
938	Thomas	Abernathy	m

Casts:

pid	**mid**	**role**
938	52432	Dancer
938	52432	Dancer

Com: 1.0 Con: 0.42 Real: 1.0

Figure 5.14: The samples generated by the SIQ technique versus the Olston technique for a query with a join [**Query 4**].

Meanwhile, for our Pig translation, we alter the query slightly to include a filter that says `year == ’2000’`. Due to irregular delimiters, the `Movie.csv` file is parsed in such a way that the `year` column is parsed as a string. Thus, we could not include the numeric predicate. The Olston technique achieves a higher conciseness than the SIQ technique, but is unable to achieve a perfect completeness score as SIQ did.

This case-by-case study of the above six different queries shows us the subtle interactions between completeness and conciseness. It also shows us that, even for single queries, the SIQ algorithm is able to generate datasets of similar and sometimes even better quality than the Olston technique. The SIQ algorithm always prioritizes completeness over conciseness. In contrast, the Olston algorithm sometimes achieves higher completeness or conciseness, but at the cost of realism and by violating key constraints.

Query | **Toy Database Generated**

```
SELECT    md.did, count(*)
FROM      Movie m, Movie_Directors md
WHERE     m.id = md.mid
GROUP BY  md.did
```

Movie:

id	name	year
21725	Asfaltlggerne	1897
45989	Brandvsnet rykker ud	1897

Movie_Directors:

did	mid
22604	21725
22604	45989

Com: 1.0 Con: 0.625 Real: 1.0

```
m= LOAD 'Movie.csv' USING PigStorage(',')
         AS (id, name, year);
md= LOAD 'MDs.csv' USING PigStorage(',')
         AS (did, mid);
j = JOIN m BY id, md BY mid;
g = GROUP j BY (did);
o = FOREACH g GENERATE group, COUNT(j);
ILLUSTRATE o;
```

Movie:

id	name	year
227046	Nakhet tqveni sakhe	1908

Movie_Directors:

did	mid
1671	227046
1671	124361
1671	227046

Com: 1.0 Con: 0.42 Real: 1.0

Figure 5.15: The samples generated by the SIQ technique versus the Olston technique for a query with a join and a `GROUP BY` [**Query 5**].

5.7.2 How Knowing the Future Can Help

In this experiment, we isolate the challenge of not knowing the future. The goal of this experiment is to demonstrate that, indeed, knowing the user's target query a priori can help us generate a better sample dataset versus generating a dataset based only on the user's current query.

For this experiment, we test for the following scenario. Suppose the following is the user's target query:

```
SELECT d.fname, d.lname, m.name, m.year
FROM movie m, movie_director md, director d, genre g
WHERE m.id = md.mid AND md.did = d.id AND
```

Query	Toy Database Generated

```
SELECT     d.fname, d.lname, count(*)
FROM       Directors , Movie_Directors md,
           Movie m
WHERE      d.id = md.did AND
           md.did = m.id AND
           m.year > 2000
GROUP BY   d.fname, d.lname
```

Movie:

id	name	year
93272	Dreaming	1990
323029	Taggart: fire, burn	2002

Movie_Directors:

did	mid
1167	93272
1167	323029

Director:

id	fname	lname
1167	Mike (I)	Alexander

Com: 1.0 Con: 0.71 Real: 1.0

```
d = LOAD ‘Dir.csv’ USING PigStorage(’,’)
        AS (id, fname, lname);
md= LOAD ‘MDs.csv’ USING PigStorage(‘,’)
        AS (did, mid);
m= LOAD ‘Movie.csv’ USING PigStorage(‘,’)
        AS (id, name, year);
j1 = JOIN d BY id, md BY did;
j2 = JOIN j1 BY mid, m by id;
f = FILTER j2 BY year == ’2000’;
g = GROUP f BY (fname, lname);
o = FOREACH g generate group, COUNT(f);
illustrate o;
```

Movie:

id	name	year
220510	Move On	1903

Movie_Directors:

did	mid
44	220510

Director:

id	fname	lname
44	A.C.	Abadie

Com: 0.79 Con: 1.0 Real: 1.0

Figure 5.16: The samples generated by the SIQ technique versus the Olston technique for a query that has a three-way join and a `GROUP BY` [**Query 6**].

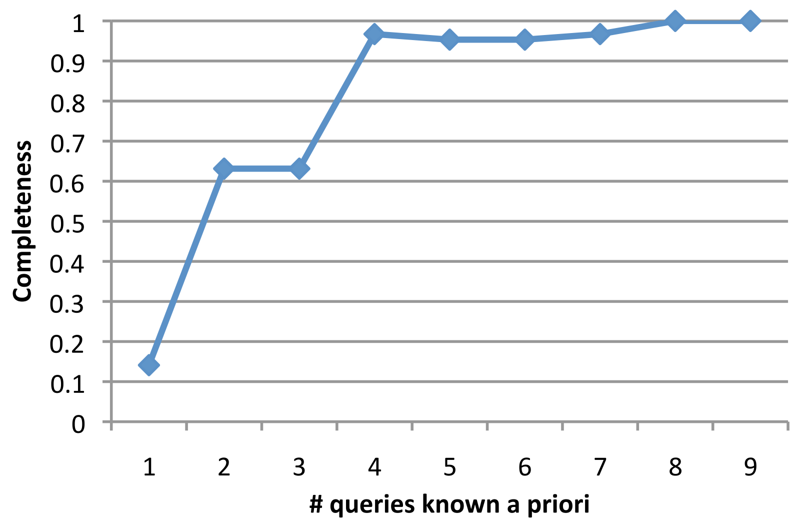


Figure 5.17: Completeness increases as we learn more about the user's queries in the session.

```
  g.mid = m.id AND g.genre = 'Film-Noir'
```

Suppose the user writes a simple `SELECT * FROM movie` query as her first query. Then, she adds the additional tables to the `FROM` clause, one table at a time. She proceeds to do the same with the `WHERE` clause (one predicate at a time), and as her final step she changes the `SELECT` clause to its final form. In total, there are nine queries in the query session.

Figures 5.17 and 5.18 plot the completeness and conciseness, respectively, as we increase the number of queries in the session that we generate the toy database for. We see that there is a tradeoff between completeness and conciseness when generating samples to support every query of the session. Namely, as we increase the number of queries in the session that we illustrate, the completeness score increases (Figure 5.17). However, we also see that if we illustrate the whole query session at once the conciseness decreases (Figure 5.18).

5.7.3 Tradeoff of Coherency versus Completeness and Conciseness

In our third experiment, we compare using SIQ that generates a single toy database for a whole session, to the Olston technique that generates a toy database at every step of the query session. We do this for ten randomly selected query sessions, and report the

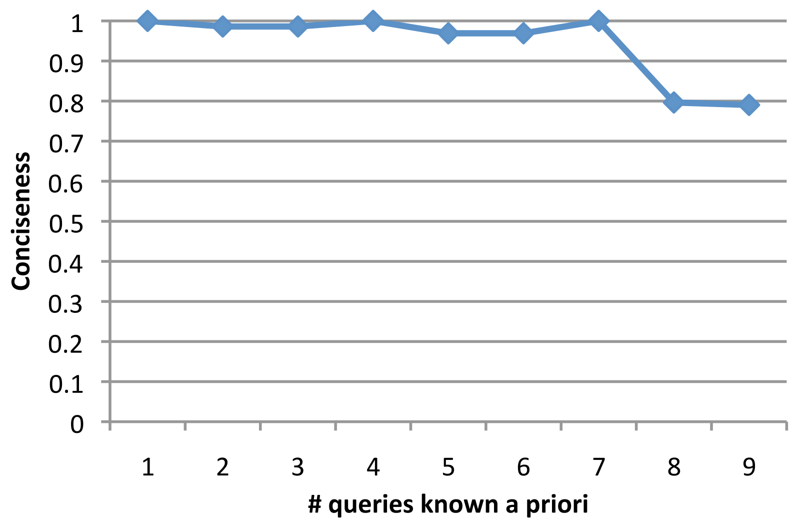


Figure 5.18: Conciseness decreases if we try to illustrate too many queries in the session.

completeness, conciseness and coherency scores.

For the SIQ part of the comparison, we first generate a toy database for a workload consisting of 500 sessions. Then we measure the quality of this single toy database for the ten random test sessions (which do not appear in the workload that SIQ used to generate the sample) and report the average completeness and conciseness. Note that with SIQ, in this experiment, the coherency score is always 1.0. Figures 5.19 and 5.20 report the results. The x-axis varies the number of tuples we allow in the toy database, and the y-axis reports the completeness and conciseness, respectively. For example, we see that when $k = 10$, SIQ achieves a completeness score of 0.66, and a conciseness score of 0.82.

For the Olston part of the comparison, we generate a toy database per query in the session. The first row in Figure 5.21 reports the average completeness, conciseness, and coherency scores for the Olston technique. The second row summarizes the results from Figures 5.19 and 5.20 for SIQ when $k = 10$.

From these results, we see that the conciseness and completeness scores decrease a non-trivial amount in order to achieve full coherency. It would be interesting to see if we can generate toy databases with better conciseness and completeness when allowing repairs, at

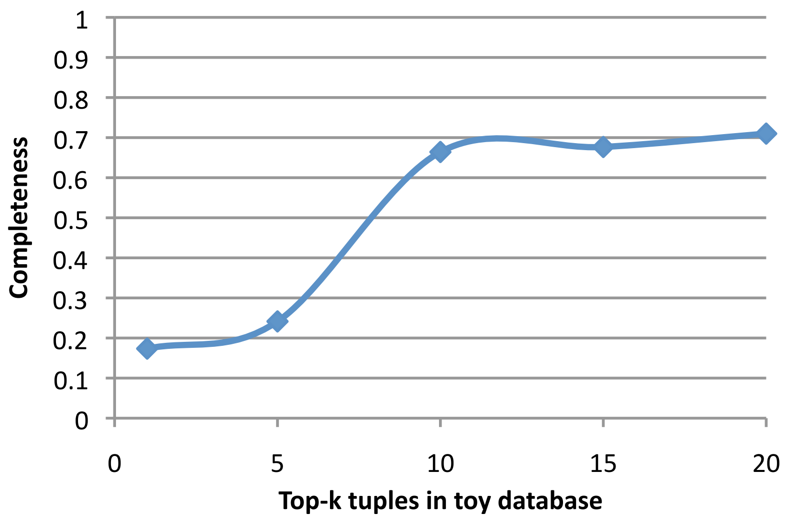


Figure 5.19: Average completeness scores for ten random sessions for SIQ-generated toy database. Coherency is 1.0.

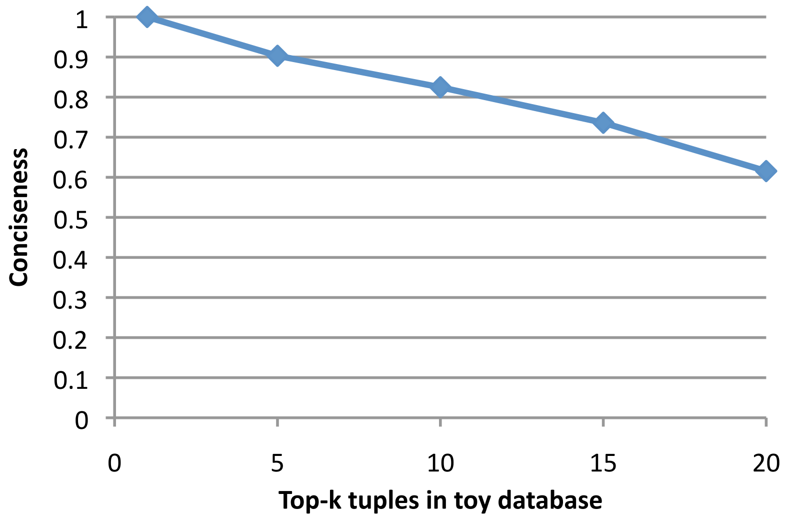


Figure 5.20: Average conciseness scores for ten random sessions for SIQ-generated toy database. Coherency is 1.0.

Technique	**Completeness**	**Conciseness**	**Coherency**
Olston	0.98	0.79	0.28
SIQ (top 10)	0.66	0.82	1.0

Figure 5.21: Average completeness, conciseness and coherency scores for the ten random sessions.

the price of coherency. However, we leave this for future work.

5.7.4 Size of Toy Database as Workload Grows

In this section, we investigate how the size of the toy database increases as we increase the number of queries in the workload. We compare two techniques. The first is the SIQ algorithm whose core algorithm is described in Algorithm 6. The second algorithm is the union-based algorithm that we discussed in Section 5.2. Remember, that we generate a dataset per query and take the union across all the queries in the workload. To generate the dataset per query, we actually use the SIQ algorithm itself (instead of the Olston algorithm), and because we have shown in Section 5.7.1 that SIQ is similar or better than the Olston technique for single queries, we believe that this is a fair comparison.

Figure 5.22 summarizes the results of this experiment. The x-axis (log-scale) shows the number of queries in the workload (i.e., the number of queries that the toy database illustrates). The y-axis reports the average number of tuples per table in the toy database. We chose to use this measure, instead of conciseness for this experiment, because we wanted a measure that is independent of queries and query sessions. We see from the figure that the union-based algorithm generates toy databases that are an order of magnitude larger than the SIQ algorithm's toy databases. For example, for 512 queries, SIQ generates a toy database consisting of an average of 7.75 tuples per table, whereas the union based algorithm generates 85 times more tuples (665 tuples).

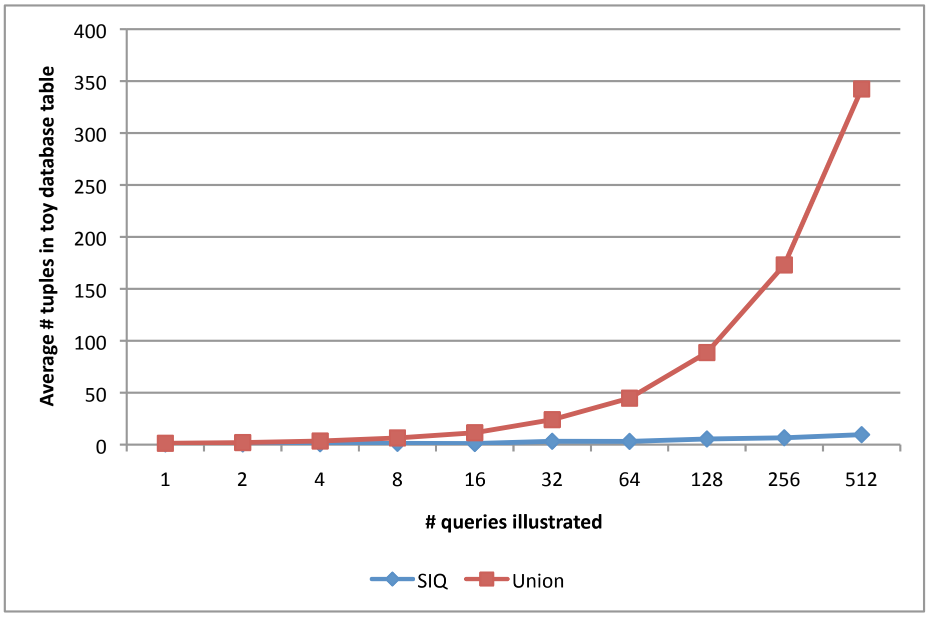


Figure 5.22: Size of toy database generated by SIQ algorithm versus union-based algorithm.

5.8 Conclusion

In this chapter, we examined the problem of query debugging, which is currently a slow and cumbersome process. Inspired by users who construct their own toy database to work with during the query debugging process, we designed and built the SIQ system for automating toy database construction. Given a query log of past query sessions, SIQ's goal is to generate a toy database that is small, and illustrative of these past sessions. We evaluated SIQ and found that it is able to generate samples that are more effective than simply using an existing technique which generates a toy database per query in a session, and generates significantly smaller toy databases than a union-based technique that generates a sample per past query and takes the union of all these samples.

Chapter 6
FUTURE DIRECTIONS

In this chapter, we discuss areas for future work. We organize the discussion around possible extensions for each of the three different projects.

6.1 Potential Directions for SnipSuggest

SnipSuggest employs a fairly generalizable algorithm for selecting its recommendations. As such the recommendation technique could be applied in different contexts. First, it could be easily extended to support other querying languages, including visual query building tools. Though SnipSuggest's original goal was to help those struggling with SQL concepts, it proved to be also helpful to users who are unfamiliar with the underlying schema. Therefore, we believe that the technique could also be used for programming; the system could recommend relevant parts of the application programming interface (API) to the user. This new problem would pose additional challenges, however. For example, with SQL, the system knows the semantics of each operator, whereas this is not the case with many imperative languages (such as Java). Second, if the recommendations are over an API that is still evolving, this will create a whole new set of challenges (whereas schemas tend to be more persistent).

One drawback of the SnipSuggest system is that it relies on an extensive query log. If the query log is small or even non-existent, the recommendation techniques that use only popularity (while ignoring the context), or that use only the schema of the database may prove to be more helpful. As such, a useful extension to SnipSuggest would be to add support to self-adjust among the different techniques as the query log evolves over time.

A contrasting problem is if the query log is too large and too diverse. This problem may occur when there is a single shared database, and many different groups of people are writing queries over it. If there are too many trends and patterns in the log, then maintaining high recommendation accuracy is challenging. In such a setting, a possible direction to pursue would be to cluster together users with similar querying patterns. Therefore, when a new user arrives, we can quickly determine which cluster the user belongs to, and thus offer better recommendations. Similarly, querying patterns may evolve over time. For example, tables that were explored together in the past may no longer be of interest to current users. Even schemas occasionally are modified, thus invalidating past queries. Therefore, techniques for gradually expiring queries over time may prove necessary in long-running settings.

6.2 Potential Directions for PerfXplain

Like SnipSuggest, PerfXplain's language for specifying performance queries and it's algorithm for generating explanations are general. As such, the technique could be applied to different settings. For example, with the PerfXplain technique, one could debug the performance of SQL queries, Java programs, and so on. The key challenge to these extensions would be in picking an effective set of features, that are both predictive of the duration of a query or program, but also not too specific.

Another interesting perspective to explore with PerfXplain is the relationship between optimizations and explanations. In some cases, the user does not care to understand why a job was slower than another job. They simply want the job to execute quickly. One interesting question here is how do we translate an explanation into an optimization? However, not all explanations are actionable. For example, a job may have been slower because it was executed when the cluster was busy or because the input file was large. So, how can we distinguish between explanations that are actionable versus not? More generally, is it possible to build an automatic optimization system on top of an effective explanation system such as PerfXplain?

Sometimes the explanations generated by PerfXplain need explaining. For example, saying that despite having different input sizes, two jobs had a similar duration because the block size was large may not be helpful to a user who does not understand what a block size is. This calls for a system with more support for such non-experts. Perhaps associated with each parameter should be a short explanation of what the parameter means (written by experts), typical values seen for this parameter, as well as how to change its value (if possible). Such an integrated system would require more user-effort to design and build, but would probably prove more helpful to users in the long run. It could even be constructed over time. Namely, the system could support a way to request additional explanations, and so when a user sees an explanation that they do not understand, they can trigger such a request.

PerfXplain currently supports queries about pairs of jobs. A natural question that arises from this is: how do we support questions about single jobs? An example is a query that asks why a job was fast or slow, without giving reference to a second job. In such settings, perhaps PerfXplain could find such a point of reference (a similar job that performed differently), and then utilize its existing algorithm for then generating the explanations. There are several challenges here. First, without another job to compare against, how does a user specify fast or slow? Namely, when there is another job in the query, the user can say that they expected the job to be faster than the other job or slower than the other job. Does the user now need to specify exact values? Second, if we do choose to automatically identify the job of reference - how do we pick it among multiple candidates? Or perhaps it makes more sense to present all candidates to the user and have him or her pick it?

6.3 Potential Directions for SIQ

In the current SIQ system, we support two approaches to repairing a toy database: extending and regenerating. There are however many more options that should be explored and compared. A simple one is to allow the user to modify the sample by adding tuples, deleting tuples or editing existing tuples. Another possible technique is to find a sample that is a minimal edit from the existing one. This is similar to the extending technique, except that we are now also allowed to delete tuples from the sample. Another interesting question is when to repair the toy database. Currently, we repair when the user requests a repair, however it may be interesting to investigate different techniques for automatically detecting a good time to repair the sample.

In this work, we ruled out the option of utilizing fake tuples because we were concerned about confusing the user. For example, a distracting scenario is one in which the query returns a result over the sample dataset but no result when executed over the full dataset.

However, by allowing fake data, we can generate samples that are smaller or more complete. This leads us to several interesting challenges including: how do we incorporate fake tuples into our current algorithm, when do we choose to use fake data versus real data, and how do we communicate to the user which data is fake and which is real, if we communicate this at all?

The SIQ system improves the query debugging process by reducing the time it takes to execute the query in each iteration, and the time it takes for the user to examine and understand the results. However, it still conforms to the existing query debugging model, where the user modifies the query in every iteration, executes the query over the dataset, examines the output, and returns to editing the query. An exciting direction to pursue would be to extend SIQ to provide a more seamless interaction between the query and the sample data. Why limit the user to modifying the query by only interacting with the query text itself? The user should be able to specify what they want by selecting tuples that they want in their output, as well as tuples that they want eliminated. There are several challenges. First, what is the space of hints that should be supported? Examples of hints are 'I want this tuple', 'I do not want this tuple', 'I want more tuples like this', and so on. Second, how should the user communicate such hints to the system? Third, such hints may be ambiguous as to what change is required to the query. Namely, there may be many different edits that are compatible with the hints. Therefore, the system would need a way to help the user navigate through this space of options.

Chapter 7
CONCLUSION

The goal of this thesis work was to make database systems easier to use for non-expert database users. We focused on the challenges that users face repeatedly, every time he or she wishes to ask a question over the data.

We investigated three different challenges that are commonly faced by users. First, we addressed the problem of query composition. We wanted to make writing SQL queries easier for those who are either unfamiliar with SQL or the underlying database schema. As such, we designed and built SnipSuggest, a smart autocomplete system for SQL. SnipSuggest is a context-aware recommendation system that is able to recommend tables, views, table-valued functions in the `FROM` clause, predicates in the `WHERE` clause, columns in the `GROUP BY` clause, and more. The key idea behind SnipSuggest is to use a log of past queries to try make an informed guess about what query the user wants to compose and recommend SQL snippets from these guesses. We evaluated SnipSuggest on two different query logs, and found that it is able to generate recommendations with up to 144% higher accuracy than competing techniques.

The second challenge we addressed is that of understanding the performance of queries. Our focus in this project was on MapReduce due to its growing popularity among non-expert users, and due to the large number of configuration parameters and other factors (e.g., load conditions) that can affect the duration of a MapReduce job. In the PerfXplain work, our first contribution is a simple language that supports queries of the form 'I expected job X to be faster than job Y because it processed much less data. However, it took a similar amount of time. Why?' The second contribution of the work is an algorithm that is able to automatically generate explanations to such performance queries. To this end, we also defined the notion of a performance explanation as well as three metrics to measure the quality of an explanation (precision, generality, and relevance). Finally, we evaluated PerfXplain on a log of MapReduce jobs executed on Amazon EC2. We found that PerfXplain is able to generate high-quality explanations, compared to two non-trivial algorithms that we also implemented. It also offered a good trade-off between the precision and generality of the explanations it generated.

Finally, the third challenge we investigated was that of query debugging. Currently, query debugging is a slow process that many users struggle with. It is a slow, iterative process where each iteration involves writing or modifying a SQL query, waiting for the current query to execute over the large dataset, and examining the, often large, output of the query to understand its effects before further refining the query. To ease this process, some users construct a sample database with which they interact until they have finished constructing their query, at which point they execute the query over the full dataset. However, constructing a sample database that is helpful in illustrating the semantics of a query can be challenging. To address this problem, we design and implement the SIQ system, which can automatically generate a sample database that is illustrative of queries in a workload. We show that SIQ is able to generate samples that are of similar or higher quality than existing state-of-the-art techniques for single queries. More importantly, we show that it is able to generate significantly smaller samples than a naïve extension of existing techniques, for varying workload sizes. We also explore two different techniques for repairing a data sample when it is no longer sufficient for the user's current query.

BIBLIOGRAPHY

[1] Amazon EC2. `http://aws.amazon.com/ec2/`.

[2] Amazon web services. `http://aws.amazon.com`.

[3] Facebook. `http://www.facebook.com`.

[4] Ganglia Monitoring System. `http://www.ganglia.sourceforge.net`.

[5] Google fusion tables. `http://tables.googlelabs.com`.

[6] Large Synoptic Survey Telescope. `http://www.lsst.org/`.

[7] MySQL Query Analyzer. `http://www.mysql.com/products/enterprise/query.html`.

[8] PostgreSQL Tuning Wizard. `http://pgfoundry.org/projects/pgtune`.

[9] Salesforce.com. `http://www.salesforce.com/`.

[10] SkyServer. `http://cas.sdss.org/dr6/en/`.

[11] SqlShare. `http://sqlshare.cac.washington.edu`.

[12] The Earth Microbiome Project. `http://www.earthmicrobiome.org/`.

[13] Tuning Your PostgreSQL Server. `http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server`.

[14] Twitter. `http://www.twitter.com`.

[15] Twitter Blog: #numbers. `http://blog.twitter.com/2011/03/numbers.html`.

[16] Venture Development Corporation (VDC). `http://www.vdcresearch.com/`.

[17] Vmware. `http://www.vmware.com/`.

[18] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and Aristides Gionis. Automated ranking of database query results. In *Proc. of the First CIDR Conf.*, pages 888–899, 2003.

[19] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server demo. In *Proceedings of the 31st SIGMOD International Conference on Management of Data*, pages 930–932, 2005.

[20] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th VLDB Conference*, pages 1110–1121, 2004.

[21] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL Databases. In *Proceedings of the 26th VLDB Conference*, pages 496–505, 2000.

[22] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th VLDB Conference*, pages 496–505, 2000.

[23] Arvind Arasu, Raghav Kaushik, and Jian Li. Data generation using declarative constraints. In *Proceedings of the 37th SIGMOD International Conference on Management of Data*, 2011.

[24] Shivnath Babu. Towards Automatic Optimization of MapReduce Programs. In *Proc. of the 1st ACM symposium on Cloud computing (SOCC)*, pages 137–142, 2010.

[25] R. Baeza-Yates and B. Riberio-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing, Boston, MA, 1999.

[26] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *Proceedings of the 30th VLDB Conference*, pages 564–575, 2004.

[27] BaseNow. Database front-end applications. About SQL Query Builder. `http://www.basenow.com/help/About_SQL_Query_Builder.asp`.

[28] Donna Becker and Paul A. Barsch. Strike it Rich: Application Tuning Helps Companies Save Money through Query Optimization. Teradata magazine online. `http://www.teradata.com/tdmo/v07n04/FactsAndFun/Services/StrikeItRich.aspx`.

[29] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of the 18th International Conference on Data Engineering (ICDE)*, pages 431–440, 2002.

[30] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: generating query-aware test databases. In *Proceedings of the 33rd SIGMOD International Conference on Management of Data*, pages 341–352, New York, NY, USA, 2007. ACM.

[31] Andrei Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

[32] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st VLDB Conference*, pages 1097–1107. VLDB Endowment, 2005.

[33] Peter Buneman, Sanjeev Khanna, and Wang chiew Tan. Why and where: A characterization of data provenance. In *In ICDT*, pages 316–330. Springer, 2001.

[34] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proc. SIGCHI*, 1991.

[35] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *J. of Visual Languages & Computing*, 8(2):215–260, 1997.

[36] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[37] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. Query recommendations for interactive database exploration. In *SSDBM 2009*, pages 3–18.

[38] Surajit Chaudhuri and Vivek Narasayya. AutoAdmin What-if Index Analysis Utility. *SIGMOD Rec.*, 27:367–378, June 1998.

[39] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd VLDB Conference*, pages 3–14, 2007.

[40] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the 23rd VLDB Conference*, pages 146–155, 1997.

[41] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1:379–474, April 2009.

[42] Coscripter. `http://coscripter.research.ibm.com/`.

[43] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic SQL tuning in oracle 10g. In *Proceedings of the 30th VLDB Conference*, pages 1098–1109, 2004.

[44] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th OSDI Symp.*, pages 137–149, 2004.

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[46] Mark Derthick, John Kolojejchick, and Steven F. Roth. An interactive visual query environment for exploring data. In *Proc. UIST*, pages 189–198, 1997.

[47] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proceedings of the 36th VLDB Conference*, 3:515–529, September 2010.

[48] Doug Downey, Susan Dumais, Dan Liebling, and Eric Horvitz. Understanding the relationship between searchers' queries and information goals. In *CIKM*, pages 449–458, 2008.

[49] Doug Downey, Susan T. Dumais, and Eric Horvitz. Models of searching and browsing: Languages, studies, and application. In *IJCAI*, 2007.

[50] Songyun Duan, Shivnath Babu, and Kamesh Munagala. Fa: A System for Automating Failure Diagnosis. *Proc. of the 25th ICDE Conf.*, pages 1012 –1023, 2009.

[51] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning Database Configuration Parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.

[52] Magdalini Eirinaki and Michalis Vazirgiannis. Web mining for web personalization. *ACM Trans. Internet Technol.*, 3(1):1–27, February 2003.

[53] My Excite. `http://www.excite.com/`.

[54] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM*, 45:314–318, 1998.

[55] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy H. Katz, and David A. Patterson. Statistics-driven Workload Modeling for the Cloud. In *SMDB*, pages 87–92, 2010.

[56] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A Case for Machine Learning to Optimize Multicore Performance. In *HotPar*, 2009.

[57] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *HotPar*, 2009.

[58] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting Multiple Performance Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proc. of the 25th ICDE Conf.*, pages 592–603, 2009.

[59] Amrchana Ganapthi, Harumi Kuno, Umeshwar Daval, Janet Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple performance metrics for queries: Better decisions enabled by machine learning. In *Proc. of the 25th ICDE Conf.*, 2009.

[60] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the 35th VLDB Conference*, 2:1414–1425, August 2009.

[61] Hector Gonzalez, Jiawei Han, Xiaolei Li, and Diego Klabjan. Warehousing and analyzing massive rfid data sets. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 83–, Washington, DC, USA, 2006. IEEE Computer Society.

[62] Google Mashup Editor. `http://code.google.com/gme/`.

[63] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th ACM Symposium on Principles of Database Systems*, pages 31–40, New York, NY, USA, 2007. ACM.

[64] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *Proceedings of the 29th SIGMOD International Conference on Management of Data*, pages 16–27, New York, NY, USA, 2003. ACM.

[65] Hadoop. `http://hadoop.apache.org/`.

[66] Alon Halevy, Michael Franklin, and David Maier. Principles of dataspace systems. In *Proceedings of the 25th ACM Symposium on Principles of Database Systems*, pages 1–9, New York, NY, USA, 2006. ACM.

[67] Daniel Halperin. *Simplifying the Configuration of 802.11 Wireless Networks with Effective SNR*. PhD thesis, University of Washington, 2012.

[68] Jeffrey Heer, Maneesh Agrawala, and Wesley Willett. Generalized selection via interactive query relaxation. In *Proc. SIGCHI*, pages 959–968, 2008.

[69] Herodotos Herodotou and Shivnath Babu. Xplus: a SQL-Tuning-Aware Query Optimizer. *Proceedings of the 36th VLDB Conference*, 3:1149–1160, 2010.

[70] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of the Fifth CIDR Conf.*, 2011.

[71] Melanie Herschel, Mauricio A. Hernández, and Wang-Chiew Tan. Artemis: a system for analyzing missing answers. *Proceedings of the 35th VLDB Conference*, 2:1550–1553, August 2009.

[72] Hive. `http://hadoop.apache.org/hive/`.

[73] Bill Howe and Garret Cole. SQL Is Dead; Long Live SQL: Lightweight Query Services for Ad Hoc Research Data. In *4th Microsoft eScience Workshop*, 2010.

[74] Vagelis Hristidis and Yannis Papakonstantinou. Discover: keyword search in relational databases. In *Proceedings of the 28th VLDB Conference*, pages 670–681. VLDB Endowment, 2002.

[75] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the 34th VLDB Conference*, 1:736–747, August 2008.

[76] IBM. Cognos software. `http://cognos.com/`.

[77] IBM. DB2 Performance Tuning using the DB2 Configuration Advisor. `http://www.ibm.com/developerworks/data/library/techarticle/dm-0605shastry/index.html`.

[78] IBM. IBM DB2. `http://www.ibm.com/software/data/db2/`.

[79] IBM Lotus Mashups. `http://www.ibm.com/software/lotus/products/mashups/`.

[80] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. here are my Queries. where are my Results? In *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[81] Tomasz Imieliński and Witold Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, September 1984.

[82] Incorporated Research Institutions for Seismology. `http://www.iris.edu`.

[83] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *Proceedings of the 33rd SIGMOD International Conference on Management of Data*, pages 13–24, New York, NY, USA, 2007. ACM.

[84] Eaman Jahani, Michael Cafarella, and Christopher Re. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.

[85] Magesh Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *Proceedings of the 34th VLDB Conference*, 1:695–709, August 2008.

[86] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3:472–483, 2010.

[87] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. PerfXplain: debugging MapReduce job performance. *Proceedings of the 38th VLDB Conference*, 5(7):598–609, March 2012.

[88] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. SnipSuggest: context-aware autocompletion for SQL. *Proceedings of the 36th VLDB Conference*, 4:22–33, October 2010.

[89] Gueorgi Kossinets and Duncan J. Watts. Empirical analysis of an evolving social network. *Science*, 311(5757):88–90, 2006.

[90] Georgia Koutrika, Alkis Simitsis, and Yannis Ioannidis. Explaining structured queries in natural language. In *Proc. of the 26th ICDE Conf.*, 2010.

[91] YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. In *Proceedings of the 22nd international conference on Scientific and statistical database management*, SSDBM'10, pages 132–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[92] Tessa Lau and Eric Horvitz. Patterns of search: analyzing and modeling web query refinement. In *Proc. UM*, pages 119–128, 1999.

[93] Guoliang Li, Ju Fan, Hao Wu, Jiannan Wang, and Jianhua Feng. Dbease: Making databases user-friendly and easily accessible. In *Proc. of the Fifth CIDR Conf.*, 2011.

[94] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Law, J. Myllymaki, and K. Wenger. Devise: Integrated querying and visual exploration of large datasets. In *Proceedings of the 23rd SIGMOD International Conference on Management of Data*, pages 301–312, 1997.

[95] Eric Lo, Nick Cheng, and Wing-Kai Hon. Generating databases for query workloads. *Proceedings of the 36th VLDB Conference*, 3(1-2):848–859, September 2010.

[96] Heikki Mannila and Kari-Jouko Rih. Automatic generation of test data for relational queries. *J. Comput. Syst. Sci.*, pages 240–258, 1989.

[97] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Y. Halpern, Christoph Koch, Katherine F. Moore, and Dan Suciu. Causality in databases. *Data Engineering Bulletin*, 2010.

[98] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. Why so? or why no? functional causality for explaining query answers. *CoRR*, abs/0912.5340, 2009.

[99] Microsoft. Microsoft SQL Server. `http://www.microsoft.com/sqlserver/`.

[100] Microsoft Popfly. `http://www.popfly.com/`.

[101] MicroStrategy 9. Microstrategy. `http://www.microstrategy.com)`.

[102] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 36th SIGMOD International Conference on Management of Data*, pages 507–518, 2010.

[103] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the Progress of MapReduce Pipelines. In *Proc. of the 26th ICDE Conf.*, pages 681–684, 2010.

[104] Arnab Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *Proceedings of the 33rd SIGMOD International Conference on Management of Data*, pages 1156–1158, 2007.

[105] Arnab Nandi and H. V. Jagadish. Effective phrase prediction. In *Proceedings of the 33rd VLDB Conference*, pages 219–230, 2007.

[106] Arnab Nandi and H. V. Jagadish. Qunits: queried units for database search. In *Proc. of the Fourth CIDR Conf.*, 2009.

[107] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 245–256, New York, NY, USA, 2009. ACM.

[108] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 245–256, New York, NY, USA, 2009. ACM.

[109] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 34th SIGMOD International Conference on Management of Data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[110] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a Not-So-Foreign Language for Data Processing. In *Proceedings of the 34th SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[111] Oracle. Oracle Database. `http://www.oracle.com/`.

[112] Li Qian, Kristen LeFevre, and H. V. Jagadish. Crius: user-friendly database design. *Proceedings of the 36th VLDB Conference*, 4:81–92, November 2010.

[113] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[114] Marko Robnik-Sikonja and Igor Kononenko. An Adaptation of Relief for Attribute Estimation in Regression. In Douglas H. Fisher, editor, *Fourteenth International Conference on Machine Learning*, pages 296–304. Morgan Kaufmann, 1997.

[115] David De Roure, Carole Goble, Jiten Bhagat, Don Cruickshank, Antoon Goderis, Danius Michaelides, and David Newman. myexperiment: Defining the social virtual research environment. *eScience, IEEE International Conference on*, 0:182–189, 2008.

[116] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3:460–471, 2010.

[117] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Claudio T. Silva. Querying and re-using workflows with vstrails. In *Proceedings of the 34th SIGMOD International Conference on Management of Data*, pages 1251–1254, 2008.

[118] The Scriptome - Protocols for Manipulating Biological Data. `http://sysbio.harvard.edu/csb/resources/computational/scrptome/`.

[119] Sloan Digital Sky Survey. `http://www.sdss.org/`.

[120] Tableau Software. `http://www.tableausoftware.com/`.

[121] The PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`.

[122] Vamsidhar Thummala and Shivnath Babu. iTuned: a Tool for Configuring and Visualizing Database Parameters. In *Proceedings of the 36th SIGMOD International Conference on Management of Data*, pages 1231–1234, 2010.

[123] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[124] Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *Proceedings of the 36th SIGMOD International Conference on Management of Data*, pages 15–26, New York, NY, USA, 2010. ACM.

[125] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 535–548, New York, NY, USA, 2009. ACM.

[126] Chris Weaver. Building highly-coordinated visualizations in improvise. In *Proc, INFOVIS*, pages 159–166, 2004.

[127] Tom White. *Hadoop: The Definitive Guide. MapReduce for the Cloud* . O'Reilly Media, 2009.

[128] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[129] Yahoo Pipes. `http://pipes.yahoo.com/pipes/`.

[130] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. Wysiwyg development of data driven web applications. *Proceedings of the 34th VLDB Conference*, 1:163–175, August 2008.

[131] Xiaoyan Yang, Cecilia M. Procopiuc, and Divesh Srivastava. Summarizing relational databases. *Proceedings of the 35th VLDB Conference*, 2(1):634–645, 2009.

[132] Cong Yu and H. V. Jagadish. Schema summarization. In *Proceedings of the 32nd VLDB Conference*, pages 319–330. VLDB Endowment, 2006.

[133] Moshé M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the 1st VLDB Conference*, pages 1–24, 1975.