Learning from SQL:
Database Agnostic Workload Management

Shrainik Jain

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

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Bill Howe, Chair

Ed Lazowska, Co-chair

Dan Suciu

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Learning from SQL:
Database Agnostic Workload Management

Shrainik Jain

Chair of the Supervisory Committee:
Bill Howe
Information School

Database Management Systems largely ignore the wealth of information present in SQL query workloads. In this work, we present a vision for database agnostic workload management. We start by providing an architecture for the SQLShare platform, a database-as-a-service built for researchers with minimal database experience. We demonstrate how we used this system to collect and publish a diverse workload of hand-written SQL queries to aid database research in general and workload analytics in particular. We also provide an analysis of the SQLShare workload and using the learnings from this analysis, we present the design of Querc, a database-agnostic workload management and analytics service, describe potential applications, and show that separating workload representation from labeling tasks affords new capabilities and can outperform existing solutions for representative tasks, including workload sampling for index recommendation, user labeling for security audits, error prediction for debugging, and query runtime prediction for resource allocation.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

DBMS: Database Management System

NLP: Natural Language Processing

SQL: Structured Query Language

SDSS: Sloan Digital Sky Survey

RMSE: Root Mean Squared Error

SR: Scaled Runtime

DOP: Degree of Parallelism

RAS: Runtime Accuracy Score

SLA: Service Level Agreement

QPT: Query Plan Template

LSTM: Long short-term Memory

ACKNOWLEDGMENTS

This dissertation wouldn't be possible without the continued support and invaluable guidance from my advisors Prof. Bill Howe and Prof. Ed Lazowska. Over the course of my PhD, they have helped me become the researcher that I am now. It is rare to find not one, but two advisors who are both gifted researchers and wonderful human beings. I would have succumbed to depression and quit grad school long ago, if not for the emotional support and academic advice they provided.

I would like to thank Prof. Dan Suciu, Prof. Hanna Hajishirzi and Prof. Gasper Begus for agreeing to be a part of my committee. I am one of those rare students whose committee members agreed to meet on a Sunday just so that he could graduate on time.

My internship at Snowflake Computing played a significant role in completion of this thesis. I would like to thank my mentors Jiaqi Yan and Thierry Cruanes, for their help and guidance over the past couple of years. I have been incredibly lucky to have them as collaborators, and to get a chance to try out my research ideas in a real-world product. I would also like to thank Bradley Jiang for his feedback on my work and Marcin Zukowski for giving me the opportunity to work at Snowflake.

Coffee fuels graduate students, and I would like to give a special thanks to my close friends at UW, Nacho Cano, Aditya Vashistha, Vincent Lee, Shumo Chu, Alex Mariakakis, Ravi Karkar, Dominik Moritz, Maaz Ahmed and Naveen kr. Sharma, with whom I shared countless coffees and meals that made all my research possible.

Special shout out to my past and present roommates and dearest friends, Rishabhkumar Shukla, Madhurima Pore, Susmita Rishi and Tushar Chaturvedi. Without these people, I wouldn't have survived my darkest days.

I have always maintained that it takes a family to finish a PhD. Without the constant support

of my parents, Santosh Jain and Vijay Jain, and my brothers, Sanket Jain and Swayambhoo Jain, I would never have reached UW. I might have lost faith in myself countless times, but they never have. I truly did win the genetic lottery by being related to these geniuses.

DEDICATION

To my parents,
who now have a 100% success rate
in producing PhD offsprings

Chapter 1

INTRODUCTION

Database management systems (DBMSs) are adept at optimizing queries based on the underlying patterns in data (e.g., cardinality estimates, precomputed synopses of data to estimate the cost of physical plans etc.), however using the syntactic patterns of the query workload itself remains largely untouched. In order to build better DBMSs that make optimization decisions based on not just the underlying data, but also the constantly changing query workloads, we need to augment the existing systems with the ability to make decisions based on abstract representations learned from the SQL query workloads.

There have been research efforts that use query workloads to inform a variety of database management decisions. The applications of query workload management span the entire DBMS application stack, including workload characterization (e.g., workload classification REDWAR system for DB2 [84], workload summarization for index and view selection [17, 6], user behavior analysis [3, 76, 36, 71], and query recommendation [61, 7, 42]), admission control (e.g., identifying SQL attacks [81, 45, 51, 38]), query scheduling (e.g., query placement using WiSeDB [62], and workload forecasting [59]), and execution control (e.g., query performance modeling [41, 9], and self-tuning databases [59, 62, 63, 50]).

However, the diversity of applications has led to a diversity of solutions, each relying on specialized feature engineering. This has resulted in methods being offline, unscalable to larger query workloads, and brittle with respect to changing SQL dialects and database systems. In turn, this has led to a very limited uptake of these methods and applications in practice. In order to achieve the vision of DBMSs that learn from ever changing query workloads in an online fashion, the following challenges must be addressed:

- **Challenge 1: Lack of public query workloads.** With the exception of the Sloan Digital

Sky Survey (SDSS) [3, 76], publicly accessible query workloads are rare. Query workloads are a necessity for effective research in this area, and also to bootstrap the algorithms that learn syntactic patterns from SQL queries.

- **Challenge 2: Lack of robust techniques for workload analytics.** Workload analytics so far has been performed as a standalone, offline task, and on relatively smaller workloads. The existing techniques rely on brittle parsers and manual feature engineering. Furthermore, there is also a lack of systems that mediate patterns learned from workloads and the database management systems.

This thesis proposes solutions to the aforementioned challenges and lays down groundwork for database management systems that use not just the distribution patterns of underlying data, but also the syntactic patterns learned from the query workloads. We identify the need for automation of system management. With the growing complexity of user scenarios, data and systems configuration, it is infeasible to build all-purpose systems that requires careful tuning from database administrators and customers of every specific use case. Instead, we present a vision of DBMSs that can adapt to changing workloads by learning the "right" configuration. We notice that computer science research is ill-equipped to realize our vision because researchers do not have access to use cases found in practice. To address these challenges we present the following two systems in this thesis:

1.1 SQLShare: Database-as-a-Service to collect SQL workloads

We present SQLShare, a database-as-a-service platform targeting scientists and data scientists with minimal database experience. We used this system to collect a multi-year workload of hand-written queries, along with a detailed analysis of this workload. Our hypothesis was that relatively minor changes to the way databases are delivered can increase their use in ad hoc analysis environments. The web-based SQLShare system emphasizes easy dataset-at-a-time ingest, relaxed schemas and schema inference, easy view creation and sharing, and full SQL support. We find that these features have helped attract diverse workloads typically associated with scripts and files rather than

relational databases: complex analytics, routine processing pipelines, data publishing, and collaborative analysis. Quantitatively, these workloads are characterized by shorter dataset "lifetimes," higher query complexity, and higher data complexity. The workload suggests a need for a new class of relational systems emphasizing short-term, ad hoc analytics over engineered schemas to improve uptake of database technology. This work is also accompanied with a data-release of the SQL workload collected spanning a period of over four years.

1.2 Querc: Database agnostic workload management

Our learning from the analysis of the SQLShare workload motivated our work on more generalized, robust and scalable techniques for workload analytics, while at the same time informing us of the power of syntactic analysis of SQL queries. We present a system to support generalized SQL workload analysis and management for multi-tenant and multi-database platforms. Workload analysis applications are becoming more sophisticated to support database administration, model user behavior, audit security, and route queries, but the methods rely on specialized feature engineering, and therefore must be carefully implemented and reimplemented for each SQL dialect, database system, and application. Meanwhile, the size and complexity of workloads are increasing as systems centralize in the cloud. We model workload analysis and management tasks as variations on query labeling, and propose a system design that can support general query labeling routines across multiple applications and database backends. The design relies on the use of learned vector embeddings for SQL queries as a replacement for application-specific syntactic features, reducing custom code and allowing the use of off-the-shelf machine learning algorithms for labeling (classification). The key hypothesis, for which we provide evidence in this thesis, is that these learned features can outperform conventional feature engineering on representative machine learning tasks. We present the design of a database-agnostic workload management and analytics service, describe potential applications, and show that separating workload representation from labeling tasks affords new capabilities and can outperform existing solutions for representative tasks, including workload sampling for index recommendation, user labeling for security audits and error prediction.

1.3 Contributions

We make the following contributions:

- A description of the SQLShare Database-as-a-Service, a system designed to increase uptake of database technology for ad hoc analysis and deployed as the instrument used to collect SQL workloads.

- A new publicly available ad hoc SQL workload dataset of 24275 hand-written queries over 3891 user-uploaded tables provided by scientists and data scientists in the life, physical, and social sciences. An initial analysis of the SQLShare workload linking the features of SQLShare to specific usage patterns typically associated with scripts-and-files, along with a general characterization of these usage patterns.

- We propose Querc, a database-agnostic system for mining and managing large-scale and heterogeneous workloads. We model workload management and analysis as a set of query labeling tasks. We evaluate these algorithms on real workloads from the Snowflake Elastic Data Warehouse [20] and TPC-H [4], showing that the generic approach can improve performance over existing methods. We demonstrate that it is possible to pre-train models that generate query embeddings and use them for workload analytics on unseen query workloads.

- Resource prediction and management for distributed query compilation and execution: we propose to extend the techniques introduced in the Querc system in order to aid query optimizers in making better decisions regarding resource allocation and management at various stages of distributed query compilation and execution.

The following companion papers have been published along with this thesis:

- **SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment**: **Shrainik Jain**, Dominik Moritz, Daniel Halperin, Bill Howe, Ed Lazowska, In proceedings of the 2016

ACM SIGMOD International Conference on Management of Data. (Received "Most Reproducible Paper" award at SIGMOD 2017.)

- **Data Cleaning in the Wild: Reusable Curation Idioms from a Multi-Year SQL Workload**: **Shrainik Jain**, Bill Howe, In proceedings of the 11th International Workshop on Quality in Databases.

- **High Variety Cloud Databases**: **Shrainik Jain**, Dominik Moritz, Bill Howe, In proceedings of the 2016 IEEE Cloud Data Management Workshop.

- **Database-Agnostic Workload Management**: **Shrainik Jain**, Jiaqi Yan, Thierry Cruanes, Bill Howe. In proceedings of the 9th biennial Conference on Innovative Data Systems Research (CIDR 2019).

- **Query2Vec: An Evaluation of NLP Techniques for Generalized Workload Analytics**: **Shrainik Jain**, Bill Howe, Jiaqi Yan, Thierry Cruanes. In ArXiv e-prints (Jan. 2018) arXiv:cs.DB/1801.05613.

- **Snowtrail: Testing with Production Queries on a Cloud Database**: Jiaqi Yan, Qiuye Jin, **Shrainik Jain**, Stratis D. Viglas, Allison Lee. In proceedings of 7th International Workshop on Testing Database Systems (DBTEST 2018).

This thesis is structured as follows: Chapter 2 covers the background and related work, along with an overview of methods for textual representation learning. Chapter 3 provides an in-depth architecture of the SQLShare system, the workload collected and lessons learned from a multi-year SQL-as-a-Service experiment. Chapter 4 presents Querc, a system that enables generalized workload analytics, followed by an overview of the performance of our system on various workload analytics tasks. In chapter 5 we use the techniques presented in this thesis to predict query runtime and to aid query scheduling and resource allocation for the Snowflake data warehouse [20], which is followed by concluding remarks in Chapter 6.

Chapter 2

RELATED WORK

In this chapter we present the related work in the field of workload management and representation learning, followed by a discussion about how the contributions made in this thesis augment and fill in the gaps in current workload management approaches.

2.1 Workload Management

Workload management entails observing and administering work submitted to a DBMS in a manner that optimizes the use of resources available, while meeting service level agreement (SLA) provided to the end user. Zhang et al. [88] provide an overview and taxonomy of workload management techniques implemented by various DBMS vendors in industry and literature. They classify various workload management tasks into one of the following buckets: *workload characterization* (identify workload classes or types), *admission control* (decisions pertaining to accepting/rejecting a query for execution), *scheduling* (query routing) and *execution control* (query resource management). Decades ago, Yu et al. characterized relational workloads as a step toward designing benchmarks to evaluate alternative design tradeoffs for database systems [84]. They presented a system REDWAR that characterized workloads submitted to the DB2 system based on query complexity, presence of relations and views and runtime behavior. Ren et al. performed a workload analysis over 3 different Hadoop [1] research clusters [71]. They noted underuse Hadoop features and significant diversity in workloads application styles motivating newer tools. Singh et al. published a 5 year usage study of SDSS [76] and reasoned about why it was so successful. Their work analyzed traffic and sessions by duration, usage pattern over time and found interesting factors like site's popularity and benefits of providing a framework for ad hoc SQL.

Workload management is not limited to just the DBMS context. Alspaugh et al. [10] pub-

lished a thesis on log analysis for data visualization software, Splunk and Tableau, with the aim of identifying higher level user tasks from low level application logs. Their work identifies the key challenges in analyzing low-level user interaction logs, techniques for clustering logs based on higher level activity patterns for two visualization software, and a description of the higher level user activity identified using their log analysis techniques.

Detecting security violations Detecting security violations within DBMS is critical because all large scale applications have database backends and applications level code cannot be guaranteed to sanitize user inputs and queries. SQL attacks or security violations in DBMSs covered in the literature fall under one of the following three classes: *SQL injection*, *cross-site scripting*, and *data-centric attacks*. Most papers in this area deal with detection of SQL injections, i.e. the attacks that involve injecting inputs that are meaningful (but harmful) SQL statements. Kim et al. [45] and Ladole et al. [51] propose methods that use support vector machines (SVMs) based frameworks to classify safe versus unsafe queries. Both of these work start out by templatization of query trees, followed by carefully hand-crafted feature generation (i.e., converting an input query to query tree template and then converting the query tree into n-dimensional feature vectors that can be consumed by the SVM). Kar et al. presented SQLiDDS [38] that is yet another framework for detecting SQL injections, however there were two key differences in their approach: first was the use of document similarity scores to cluster similar SQL queries, and the second was exploiting the fact that most SQL injections occur in the WHERE clause in SQL query to improve the training time and performance of their proposed model. A more comprehensive classification and detection of SQL attacks was provided by Valeur et al. [81] with a focus on detecting not just SQL injections, but cross-site scripting and data-centric attacks as well. The outline of their approach, however, was similar to the other works in this field, i.e., parse and templatize the input query, perform a manual feature selection and feed the features to a known machine learning model to train classifiers.

Resource and performance prediction There has been a rise in employing machine learning techniques to workload management. Akdere et al. [9] presented a query performance modeling

and prediction framework based on linear regression and SVMs. They present a framework that uses carefully designed hand-crafted features that capture coarse grained plan-level information (e.g., total plan cost, cardinality estimate etc.) and finer operator level information (e.g., estimated I/O, operator selectivity etc.), to predict query performance for sub-plans (using plan-level features) and individual operators (using operator-level features). Khoshkbarforoushha et al. [41] identified that most prior workload performance modeling frameworks assume that the probability density distribution function (pdf) for runtime is available when it is actually unknown, and present a solution to this problem using Mixture Density Networks (e.g. Gaussian Mixture Model (GMM) augmented with a multilayer perceptron (MLP)), to accurately learn the underlying pdf. They perform manual feature engineering to identify the best set of features that should be extracted from query plans for their setup (TPC-H queries written for the HIVE platform). Marcus et al. introduced the WiSeDB system [62] that uses supervised learning (specially, decision trees based models) to advise DBMS for query placement, resource management and query scheduling for both batch and online workloads. Ma et al. presented QueryBot 5000 [59], a workload forecasting system built as a stepping stone towards achieving the eventual goal of building a self-driving DBMS. QueryBot 5000 extracts patterns and templates from incoming queries and clusters the workload based on arrival rate patterns followed by training forecasting models (based on an ensemble of linear regression and recurrent neural networks, or kernel regression) for each of the cluster type.

More recently, with the advent of deep learning and more specifically deep reinforcement learning (DRL), there has been a growing trend that utilizes the recent advances in DRL (e.g., Q-Learning) to aid query optimizers. Most notably, the works from Krishnan et al. [50] and Marcus et al. [63] have explored the use of DRL to learn patterns from query workloads to aid classical database tasks like join order optimization [50] and end-to-end query plan optimization [63]. While these works are still in very early stages ([50] works for Select-Project-Join blocks only and [63] explores the research challenges that need to be addressed to realize the vision of Hands-Free query optimization), they provide a good overview of the relevant deep learning techniques, challenges faced while using these techniques in a database setting and the gaps that need

to be filled by future works. It is worth noting however, that both of these visions still employ heuristics based manual feature engineering for converting raw queries to feature vectors that can be consumed by the downstream training models.

Query recommendation SQL queries written for analytical purposes have a high degree of redundancy across different user sessions. We note in our study of the Sloan Digital Sky Survey (SDSS) workloads that over 97% of 7 million queries submitted over a period of several months were duplicate. In such static schema scenarios, user tasks can often be predicted based on the current session, user query profile and historical query logs. Akbarnejad et al. presented QueRIE [7], a system which recommends queries using fragment based similarity for the SDSS workloads. The overall workflow of such recommender systems is similar to the other workload management tasks we have talked about in this section, i.e., we start from a workload of queries, write parsers to templatize and extract hand-crafted features, and feed them to downstream machine learning models for training classifiers. In a survey of query recommendation techniques for exploration, Marcel et al. [61] formalize the problem of query recommendation as follows: given a query log, a session, a user profile, a database instance, and an exploration function, predict a ranked list of possible queries that the user is most likely to submit. With the recommendation problem thus defined, the author present a comparative analysis of various query recommendation systems in the literature and identify the following exploration dimensions that these methods exploit: User profile, similarity (or distance) between sessions, and query logs. All of the surveyed approaches were found to have strict dependency on features extracted from query logs.

Compressing SQL Workloads Workload summarization involves representing each query with a set of specific features based on syntactic patterns. These patterns are typically identified with heuristics and extracted with application-specific parsers: for example, for workload compression for index selection, Chaudhuri et al. [17] (approach reused by Kolaczkowski et al. [49]) identify patterns like query type (SELECT, UPDATE, INSERT or DELETE), columns referenced, selectivity of predicates and so on. These patterns are then used to cluster similar queries using a heuristics

based distance functions and K-Mediods based clustering algorithm.

2.2 Word Embeddings and representation learning

Word embeddings were introduced by Bengio et al. [12] in 2003, however the idea of distributed representations for symbols is much older and was proposed by Hinton in 1986 [29]. Mikolov et al. demonstrated *word2vec* [65, 66], an efficient algorithm which uses negative sampling to generate distributed representations for words in a corpus. *word2vec* was later extended to finding representations for complete documents and sentences in a follow-up work, *doc2vec* [52]. Levy et al. provided a theoretical overview of why techniques like *word2vec* [55, 22, 70] work so well. Generic methods for learning representations for complex structures were introduced in [75, 11]. Representation learning for queries has been implicitly used by Iyer et al. [33] in some recent automated code summarization tasks, using a neural attention model, whereas we evaluate general query embedding strategies explicitly and explore a variety of tasks these embeddings enable. Zamani et al. [85] and Grbovic et al. [24] proposed a method to learn embeddings for natural language queries or to aid information retrieval tasks, however we consider learning embeddings for SQL queries and their applications in workload analytics. LSTMs have been used for various text encoding tasks like sentiment classification by Wang et al. [82], machine translation by Luong et al. [58] and as text autoencoders by Li et al. [56]. Our work is inspired by the success of these approaches and demonstrates their utility for SQL workload analytics.

2.3 Summary

Workload management tasks can be classified along one or more of the following dimensions: workload characterization, admission control, scheduling and execution control. The diversity of applications has led to a diversity of solutions, each relying on specialized feature engineering. For example, workload summarization for index recommendation uses the structure of join and group by operators as features [17], query recommendation may pre-process a query into fragments before making recommendations [43], and security audits may require user-defined functions to enforce particular policies [80].

All of the workload management applications listed in section 2.1 tend to execute the following generic workflow:

- Step 1: Collect query workload from historical/online logs. Use a standardized benchmark (e.g., TPC-H) if a representative workload is unavailable.
- Step 2: Write a SQL parser to templatize and extract heuristics based feature vectors for queries.
- Step 3 (Optional): Write a query plan parser to extract features from logical or physical plans.
- Step 4: Use the feature vectors from steps 2 and 3 to inform database management decisions either using heuristics based algorithms or machine learning based algorithms.
- Step 5: Evaluate the model built in the previous step on standardized benchmarks (e.g., TPC-H) or small private benchmarks and report the results.

Studying the generic workflow provided above we notice obvious gaps in prior works in this space that have limited the wider adoption of these methods in practice. This thesis attempts to address these gaps. These gaps are as follows:

- Unavailability of training workloads: most of the systems end up using benchmarks like TPC-H for training and evaluation. While TPC-H is a good benchmark for measuring raw performance metrics for execution, it lacks the diversity that real-life analytics query exhibit [36].
- Manual feature engineering: Every workload management application in the literature introduces a new hand-crafted feature engineering scheme that specifically benefits their algorithm. This feature engineering step consumes much of the real estate in every research paper in this field.

- Over reliance on brittle parsers: Manual feature engineering requires a strict dependency on SQL parsers. In the present context where most vendors are moving to cloud data warehouse offerings (e.g., Snowflake [20], BigQuery [64], Redshift [26] and the like), such parsers change frequently (over a period of 6 months, the SQL parser for the Snowflake data warehouse changed an average of 10 times per month), breaking the entire downstream pipeline for workload management.

In the following chapters, we present the architecture for systems that we built (i.e., SQLShare and Querc) that provide these missing components and enable generalized database-agnostic workload management.

Chapter 3

SQLSHARE

In this chapter, we first formalize the requirements for the database-as-a-service system and then show how we address them in SQLShare. Next we analyze the SQLShare workload to show how we fared on fulfilling these requirements over a four year period. We then look at a comparison of the SQLShare workload with the other public workloads like the Sloan Digital Sky Survey (SDSS) workload and show how SQLShare queries are more diverse and complex. Next we analyze the SQLShare query workload and present our key findings. Furthermore, we present a discussion about the non-traditional workflows generated over SQLShare. We show that by making use the of databases simpler, SQLShare enabled novice database users to build data analytic skills and focus on science and domain expertise.

Finally, we extract a set of curation idioms from a five-year corpus of hand-written SQL queries. The idioms we discover in the corpus include structural manipulation tasks (e.g., vertical and horizontal recomposition), schema manipulation tasks (e.g., column renaming and reordering), and value manipulation tasks (e.g., manual type coercion, null standardization, and arithmetic transformations). These idioms suggest that users find SQL to be an appropriate language for certain data curation tasks, but we find that applying these idioms in practice is sufficiently awkward to motivate a set of new services to help automate cleaning and curation tasks. We present these idioms, the workload from which they were derived, and the features they motivate in SQL to help automate tasks. Looking ahead, we describe a generalized idiom recommendation service that can automatically apply appropriate transformations, including cleaning and curation, on data ingest.

Table 3.1: Summary of observed requirements in science and data science environments.

Requirement	Feature	Evidence
Weakly structured data	Relaxed schemas	Casting, cleaning, integration
Derived datasets	First-class views	Deep view chains, reuse, abstraction
Collaborative sharing	User-controlled permissions	Public datasets, fine-grained sharing
Complex manipulation	Full SQL	Use of complex idioms and features
Multiple users and low operational overhead	SaaS	Broad use
Low data lifetime	Relaxed schemas	"One-pass" workloads

3.1 SQLShare Platform Features

The SQLShare [31] platform was designed, built and deployed to deliver database technology into science contexts, and, as a side effect, collect a workload dataset for use by the database research community. It had been widely observed that scientific data tended to reside in flat files and spreadsheets rather than in DBMSs. From conversations with scientists regarding their data management and data analysis requirements, we discerned a number of ways in which these requirements differed from those of the typical business applications that DBMSs had evolved to support. Our hypothesis was that uptake by scientists could be increased dramatically by adapting a DBMS to better support science requirements. These requirements are described in Table 3.1. Next we look at how each of these features were built into SQLShare.

SQLShare is a cloud-hosted data management system for science emphasizing relaxed schemas and collaborative analysis. By "relaxed schemas," we mean that data can be uploaded as is, and column types are inferred automatically from the data upon ingest rather than prescribed by users. Moreover, the interfaces are designed to accommodate the management of hundreds or thousands of datasets per user instead of a single fixed schema linked by integrity constraints. SQLShare supports a "Sea of Tables" model rather than a pre-engineered schema. In this sense, it supports

usage patterns like those of a filesystem rather than a database: Datasets can be freely created without regard to global constraints. Each dataset is a collection of typed records and has a name, but otherwise the system makes no assumptions.

SQLShare supports exploratory analysis by emphasizing the derivation and sharing of virtual datasets via relational views, and eschews destructive update at the tuple level in favor of dataset-level versioning. Tasks typically considered out of scope for relational databases, including preliminary data cleaning, timeseries analysis, and statistical analysis, are implemented by creating multiple layers of views. For example, nutrient information in an environmental sensing application may contain string-valued flags indicating missing numeric data, incorrect column names, and may comprise many separate files instead of one logical dataset. Instead of demanding that these issues be resolved prior to ingest into the system, SQLShare encourages data to be uploaded "as-is" and repaired using database features. Users may write one view to rename columns, another to replace missing values cast types, a third to integrate the files into one logical dataset, and a fourth to bin the data by time to compute an hourly average. Any of these views can be shared with collaborators as needed, and complete provenance of how the final result was constructed from raw data is available for inspection.

SQLShare was deployed and managed as a cloud-hosted service, which has been critical to its success: The backend system was deployed and supported by either zero or one developer at any time, thanks to the automatic failure handling and simplified deployment offered by the cloud providers.

The SQLShare experiment has been running in various forms since 2011, and we have attracted hundreds of science users who have run tens of thousands of queries over thousands of datasets.

3.1.1 Relaxed Schemas

Datasets are uploaded to SQLShare via the REST interface. Although we anticipated adding support for a number of different file formats, in practice we found that nearly all data was presented in some variant of row-delimited and field-delimited format, e.g. csv. Files are staged server-side and briefly analyzed to infer column types and assign default column names if necessary. Some-

what surprisingly, almost 50% of the datasets uploaded did not have column names supplied in the source file.

Once a schema is derived, the appropriate table is created in the database and the file is ingested. By staging the file server-side we ensure robustness: if ingest fails, we can retry without forcing the user to re-upload the data. To infer the format, we consider various row and column delimiter values until the first N rows can be parsed with identical column counts. To infer column types, the first N records are inspected. For each column, the most-specific type is identified. For example, if every value can be successfully cast as an integer, the type is assumed to be an integer. This prefix inspection heuristic can fail, and non-integer types may be encountered further down in the dataset. In that case, the database raises an exception, we revert the type to a string via ALTER TABLE, and the ingest continues. Besides mixed-type columns, we also tolerate non-uniform row lengths. Additional columns are created as needed to accommodate the longest row, and these columns are padded with NULL for rows that do not supply appropriate values. A total of 9% of the datasets uploaded to SQLShare made use of this feature.

Our goal with this data ingest process is to tolerate (and ideally to flag and expose) many types of data problems, including problems with the structure, types, or values. We have designed the system to ensure that we do not reject such dirty data, because many of our target users (researchers in physical, life and social sciences) have no capacity to clean, reformat, or restructure the data offline. If we force them to use scripts (or even spreadsheets) to clean the data as a preprocessing step, we are essentially asserting that SQLShare is irrelevant for their day-to-day tasks. Instead, we want to tolerate malformed data and encourage the use of SQL itself to scale and automate cleaning and restructuring tasks.

3.1.2 Data Model: Unifying Views and Tables

We illustrate the data model of SQLShare in Figure 3.1.

Views are a first-class citizen in SQLShare. Views are created in the UI (or programmatically in the REST interface) by saving a query and giving it a name. Everything in SQLShare is accomplished by writing and sharing views: Users can clean data, assess quality, standardize units,

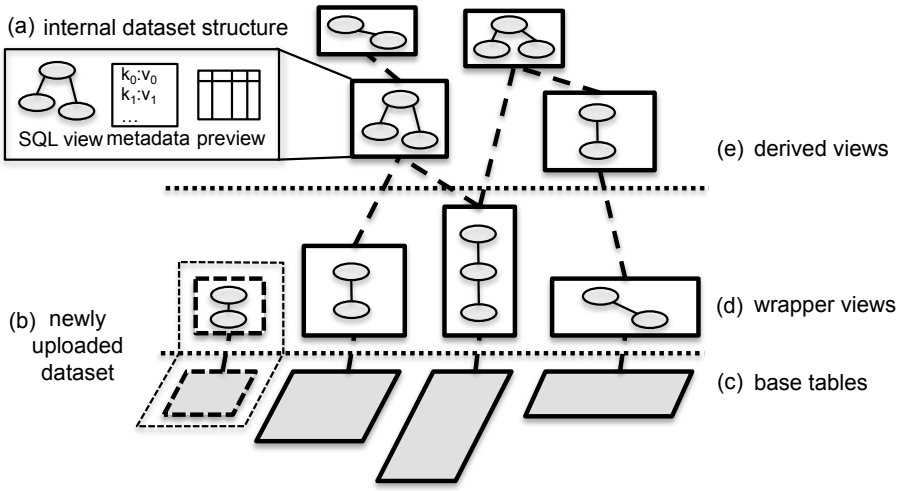


Figure 3.1: The SQLShare data model. (a) The internal structure of a dataset, consisting of a relational view, attached metadata, and a cached preview of the results. (b) A newly uploaded datasets creates both a physical base table and an initial (trivial) wrapper view. (c) The physical data are stored in base tables, but never modified directly. (d) Wrapper views are indistinguishable from other SQLShare datasets, except that they reference physical base tables. (e) Derived datasets can be created by any user. Permissions are managed in the underlying database.

integrate data from multiple sources, attach metadata, protect sensitive data, and publish results. We avoid forcing the user to use the CREATE VIEW syntax of the SQL standard, for two reasons: First, we want a view to be conceptually identical to a physical table — our design principle is "everything is a dataset." Second, the syntax proved awkward in initial tests with users.

All datasets are considered read-only; the only way to modify a dataset is by changing its view definition. Using UNION queries, a view definition can be extended with new data to simulate batch INSERTs. The advantage of this design is that provenance is maintained: an uploaded batch of data can be "uninserted" at a later date, and the substructure of the dataset as a sequence of batch inserts can be inspected and reasoned about. The disadvantage of this approach is that it prevents tuple-at-a-time updates and inserts. However, we find that a key characteristic of our target workloads is dataset-at-a-time processing, and we have not seen this design principle reported as

a weakness. The REST interface provides some convenience features for appending batches of tuples: An `append` call accepts an existing dataset name E and a newly uploaded dataset name N as input, and, if the schemas are compatible, the query definition associated with E will be rewritten as $(E)UNION(N)$. Downstream views and queries will automatically see the new data with no changes required. For some applications, it is important that the data doesn't change without the consumers' knowledge. In these cases, the user can `materialize` the dataset to create a snapshot that is distinct from the original view definition. SQLShare does not automatically materialize views to improve performance; there is an application-specific tradeoff with freshness that we have not yet explored how to optimize. We are exploring certain "safe" scenarios where we can make materialization decisions unilaterally.

Each *dataset* in SQLShare is a 3-tuple (sql, $metadata$, $preview$), where sql is a SQL query, $metadata$ consists of a short name, a long description, and a set of tags, and $preview$ is the first 100 rows of the dataset. When a user uploads a table to SQLShare, a *base table* `T` is created, along with a trivial *wrapper* query of the form `SELECT * FROM T`. This design helps unify the concept of tables and views and also provides an initial example query for novice SQL users to operate from. We find in practice that editing a simple query into an "adjacent" query is very easy for anyone in practice; only writing a complex query from scratch is difficult. The owner of the dataset is the user who created it; ownership cannot be transferred. Each dataset is associated with a set of keyword $tags$ to ease search and organization in the UI. The set of $permissions$ provides user-specific access. Users are not allowed to run DDL statements like `CREATE TABLE` etc. since that would make it difficult to automatically make a view on top of every table. Users can make a dataset public, share it with specific users, or keep it private. When sharing derived views, complex situations can arise. The semantics for determining access to a shared resource uses the concept of *ownership chains*, following the semantics of Microsoft SQL Server. If user A owns a table T, they can share a derived view $V_1(T)$ with user B even if the table T has not been shared, and user B will have access. But if user B then creates a derived $V_2(V_1(T))$ and shares it with user C, user C will encounter an error because the ownership chain $V2 \rightarrow V1 \rightarrow T$ is broken (i.e., it involves two different users, A and B.) We are exploring whether these semantics are too conservative for

our requirements, given that sharing is a first-class concept.

3.1.3 Query Processing

Queries are submitted to SQLShare primarily via the WebUI or sometimes directly through the REST API. REST server receives the query request, and assigns an identifier to the request which is sent back to the requesting client. The WebUI uses this identifier to regularly get results or check for query status. This was an obvious choice over an atomic request for queries as long running queries would reduce the requests the REST server can handle. The REST server uses the MS SQL Azure's `C#` library to run queries internally. As of now, we do not create any automatic indices, however this is a feature we might build later. Since the dataset updates are allowed only via creation of newer datasets, we can assume that the result of query wouldn't change over time. This allows us to save the preview results for each dataset and serve them instead of running the query every time the dataset is accessed. However the query needs to be actually run if the user submits a 'download results' request.

3.1.4 Architecture

The architecture of SQLShare appears in Figure Figure 3.2. The core system consists of a REST interface in front of a database system that implements the data model, query log, ingest semantics, manages long-running queries, handles exceptions, and manages authentication. The SQLShare REST interface is compatible with any relational database, but it was originally deployed using the Microsoft Azure Database (originally SQL Azure). The Microsoft Azure Database is mostly interface compatible with Microsoft SQL Server, except that it requires all tables to be associated with a clustered index. In SQLShare, we avoid exposing DDL to users and therefore create a clustered index by default on all columns in the database, in column order. The front-end UI is in no way a privileged application; it operates the REST interface like any other client. Indeed, other clients exist, in some cases built by the community. For example, the R client for SQLShare was written by a user based on their own requirements, and multiple javascript-based visualization

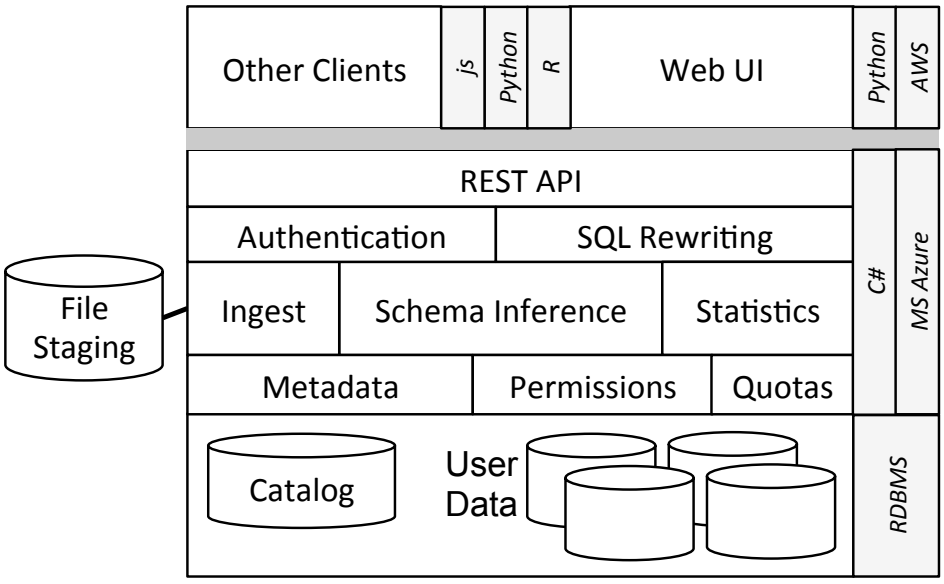


Figure 3.2: SQLShare architecture. The primary user interface is the Web UI which communicates with the REST layer for dataset ingest, modification (via views) and querying. REST server interacts with the backend database which also keeps a catalog of user queries.

interfaces have been developed.

3.1.5 Full SQL

We built SQLShare in part to faciliate access to the features of the full SQL standard, finding from our experience working with scientists that their use cases frequently required features that are not provided by simplified SQL dialects such as those found in HIVE [79] or Google Fusion Tables [23]. Specifically, window functions, unrestricted subqueries, rich support for dates and times, and set operations all appeared necessary in early requirements analysis. Since the interface is organized around a workflow of copying and pasting snippets of SQL from existing queries (a practice that in some cases may even be beneficial [72] [46]), we see evidence of users writing increasingly complex queries over time.

To support full SQL, we parse each query using a third-party standards-compliant SQL gram-

mar in ANTLR, which we modified to accommodate details of the SQL Azure database and avoid common user pitfalls. For example, when creating a view, we automatically remove any ORDER BY clause to comply with the SQL standard.

Users edit queries directly in the browser, but can access recently viewed queries to copy and paste snippets as needed. We analyze the usage of SQL features in Section 3.3.3.

3.2 Overview of the SQLShare Workload

SQLShare logs all executed queries; this log was collected to inform research on new database systems supporting ad hoc analytics over weakly structured data. With permission from the users, we released this dataset publicly for use by the database research community. To our knowledge, no other workload in the literature provides user-written SQL queries over user-uploaded datasets.

The SQLShare workload has a total of 24275 queries on 7958 datasets (including 4535 derived datasets implemented as views), authored by 591 users over period of four years. Out of 591 users, 260 are from universities (indicated by a .edu address). In addition, we have interviewed a number of our top users and are familiar with their science and their requirements. There are a total of 3891 tables with an average of 12 queries per table. Figure 3.3 shows a histogram depicting the distribution of queries per table. Most tables are either accessed just once or they are queried $>= 5$ times.

The SQLShare system is not intended for large datasets; the total volume of data presently in the system is 143.02 GB. However, users delete datasets regularly so this number doesn't represent the size of all the datasets that have ever been present in SQLShare. Indeed, based on our interactions with some users, they claimed to have developed a daily workflow of uploading data, processing it in SQL, downloading the results, and then deleting everything. Diversity rather than scale is the salient feature of the workload.

A short survey sent to all users to assess the effectiveness of SQLShare revealed that only 6 (out the 33 users who responded) felt that some other off-the-shelf database system could meet their data management needs. 18 of 33 reported that *no* other tool would work for their requirements. The remaining 9 users mentioned that non-database tools such as iPython notebooks might be

appropriate for their tasks. 23 of 33 users mentioned that "*ease of data upload & cleaning*" and "*ability to share*" was the reason they found SQLShare most helpful.

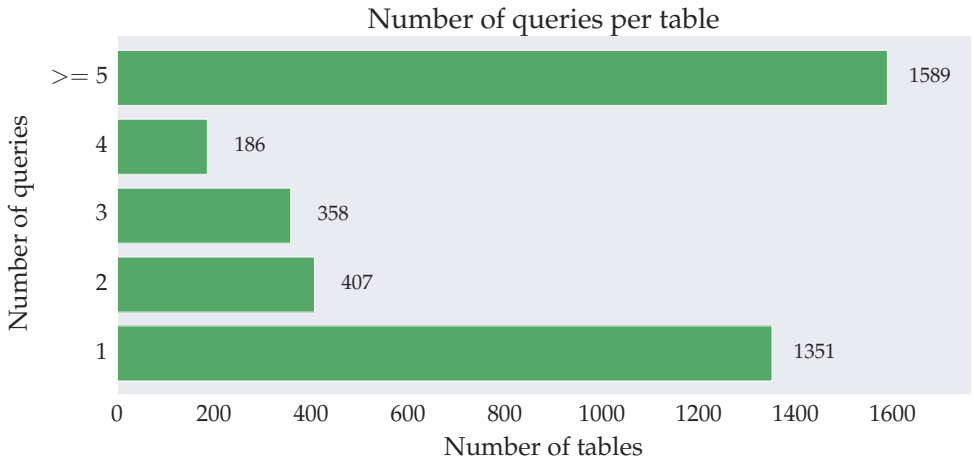


Figure 3.3: Distribution of queries per table. a) About a third of the tables accessed just once. b) Greater than a third of the tables are accessed many times, with the most common table being queried 766 times, suggesting two distinct use cases.

Extracting information from query logs To analyze the complexity and diversity of the SQLShare logs (Section 3.4), we developed a framework for extracting metrics from each query and its associated plan. The metrics of importance are query length, runtime, number & type of physical & logical operators, number & type of expression operators, tables & columns referenced and operator costs. We will use these metrics to drive the discussion the later sections.

The algorithm for extraction has 2 phases. In phase 1 each query in the SQLShare logs were sent to SQLServer, which returned the execution plan along with estimated result sizes and runtimes for each operator. The format of the execution plan is XML and is obtained by setting the `SHOWPLAN_XML` property[1]. The operator tree along with interesting properties needed for further analysis is extracted from the XML document with XPath [18]. These properties are estimated

[1] http://msdn.microsoft.com/en-us/library/ms187757.aspx

runtimes, estimated result sizes and predicates or other properties of an operator. Predicates for selections are split into clauses such that if one selection has a superset of predicates, it is more selective and filters out more tuples. Expressions are also extracted via XPath. Phase 1 extracts these properties and makes a simpler JSON plan for easier future consumption and saves it as an additional column in the query log. Figure 3.4a provides a conceptual visualization of Phase 1.

Users	591
Tables	3891
Columns	73070
Views	7958
Non-trivialViews	4535
Queries	24275

(a) Workload Metadata

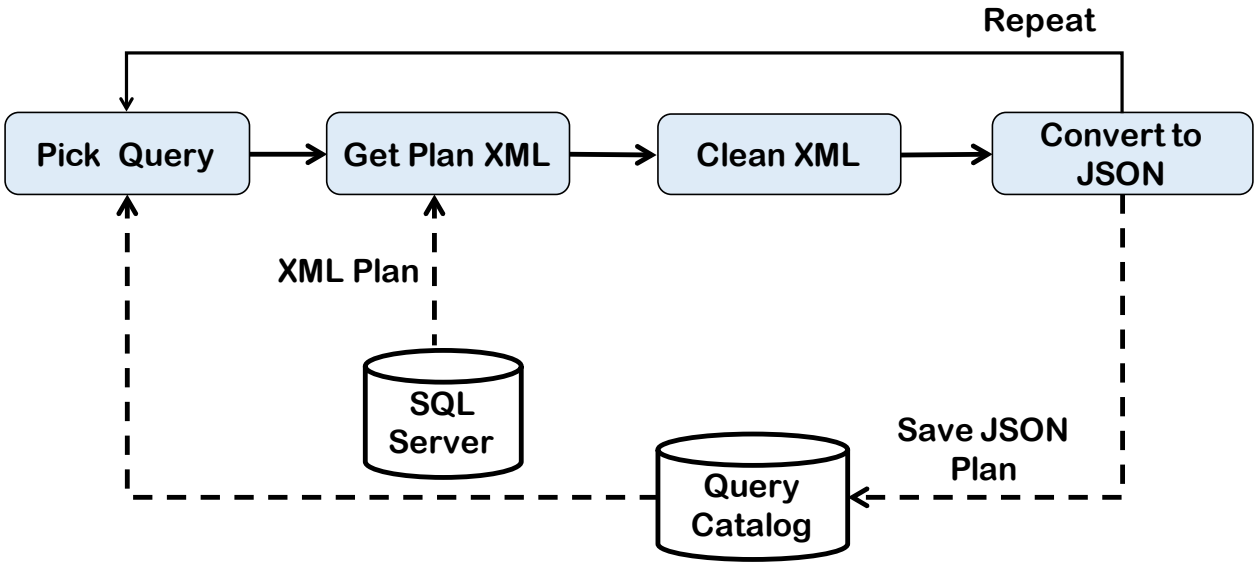
Feature	Mean Value
Length	217.32 char.
Runtime	3175.38 s.
# of Operators	18.12
# of Distinct Operators	2.71
# of Tables accessed	2.31
# of Columns accessed	16.22

(b) Query Metadata

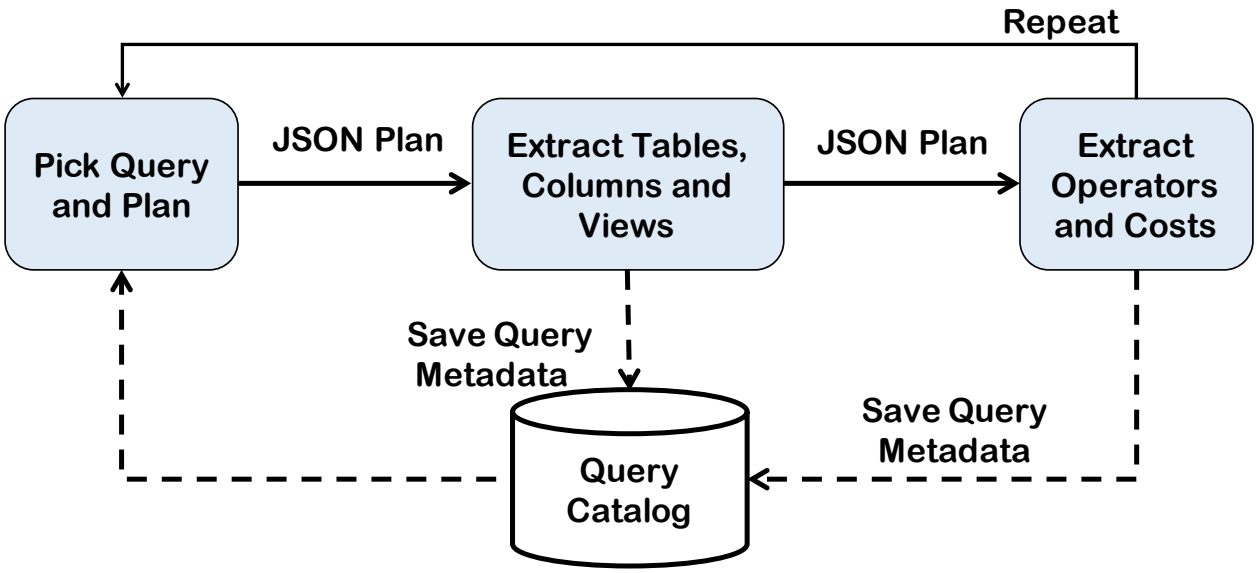
Table 3.2: Aggregate summary of SQLShare metadata. SQLShare workload has an average 12 queries per table.

Phase 2 of the algorithm goes over each query and corresponding JSON plan and extracts other important query metadata like referenced tables, columns and views per query. This metadata is aggregated into separate tables in the query catalog for further analysis of the workload. Figure 3.4b shows a flowchart for this phase of the algorithm.

Listing 1 shows a sample query and the corresponding extracted properties. Most SQL providers support the 'query explain' feature, which returns a raw query plan in the XML format, so the methodology explained in this section can be applied to other workloads as well. We implemented



(a) Phase 1 of extracting information from query log. For each query, backend SQL Server is asked to explain it and return the corresponding XML plan. The XML is then cleaned for easier parsing and the extracted information is converted to a JSON plan for easier consumption by further steps. This JSON plan is saved back to query catalog as a new column.



(b) Phase 2 of extraction methodology picks a JSON plan from previous step. From the JSON plan, the referenced tables, columns and views are extracted and saved in separate tables in the query catalog. Next the operators, expressions and corresponding costs are extracted and saved into separate tables in the catalog as well.

Figure 3.4: Workload Analysis Methodology

and bundled all functionality as a python library whose source code is available online on demand [2]. As a sample implementation for other workloads, we have provided code to perform this analysis on SDSS and TPCH [19]. The python library also implements most of analysis that we make in the sections that follow.

```
query: "SELECT * FROM incomes
           WHERE income > 500000"
physicalOp: "Clustered Index Seek"
io: 0.003125
rowSize: 31
cpu: 0.0001603
numRows: 3
filters:
  - "income GT 500000"
operator: "Clustered Index Seek"
total: 0.0032853
children: []
columns:
  incomes:
    - "name"
    - "income"
    - "position"
```

Listing 1: Extracted structure and properties from a sample query.

The total size of the query logs along with this meta data (e.g. JSON query plans) is 398 MB. This will also be made available publicly. A summary of metadata extracted from SQLShare logs

[2] https://github.com/uwescience/query-workload-analysis

is shown in table Table 3.2a and Table 3.2b. As mentioned in §3.1.2 SQLShare creates trivial views over base tables to remove the distinction between a table and a view. Hence for analysis that follows in the later section, we will only look at the non-trivial views (i.e. the ones explicitly created by users) unless otherwise specified.

3.3 Evaluation of SQLShare Features

In this section, we consider the specific features of SQLShare and analyze their effect on the usage patterns we see in the workload. Each subsection represents a key finding associated with a specific feature of SQLShare. We have conducted interviews with our most active users, who are primarily researchers in the life, earth, and social sciences. Statements about our users' backgrounds and requirements are informed by these interviews.

3.3.1 Relaxed Schemas Afford Integration

The requirement for relaxed schemas is motivated by the ubiquity of *weakly structured* data. Further, the collaborative nature of data science results in a need to frequently share intermediate results before the data has been properly organized and described. As a result, we designed SQLShare to tolerate (and even embrace) upload of weakly structured data, encouraging users to write SQL queries to repair and reorganize their data. We build evidence to support this hypothesis by searching the corpus of 4535 derived datasets (views) for specific SQL idioms that correspond to "schematization" tasks: cleaning, typecasting, and integration.

NULL injection: About 220 of the derived datasets use a `CASE` expression to replace special values with NULL.

Post hoc Column Types: We find that about 200 of derived datasets used SQL `CAST` to introduce new types on existing columns.

Vertical Recomposition: Datasets presented to SQLShare are often decomposed into multiple files that reflect the manner in which the data was collected. Rather than requiring that these files be concatenated offline or requiring that a single table be designed to store all such data, we encourage

users to "upload first, ask questions later." We found evidence of about 100 datasets that involved vertical recomposition using UNION in SQL.

Column Renaming: Datasets presented to SQLShare frequently had no column names in the source file; SQLShare automatically assigns default columns names (e.g. column1, column2 etc.) in these cases and we encourage users to write SQL to assign semantic names. We see 1996 uploaded tables (about 50%) that had at least one default-assigned column name and 1691 uploaded tables for which *all* columns names were assigned a default value. Almost 16% of datasets involve some kind of column renaming step, suggesting that users have adopted SQL as a tool for adding semantics to their data. Rejecting datasets due to incomplete column names would have clearly limited uptake.

Overall, the data suggests that relaxed schemas played an important role in many use cases, and that tolerance for weakly structured data is an important part of any data system targeting science and data science environments.

3.3.2 Views Afford Controlled Data Sharing

The view-centric data model of SQLShare (Figure 3.1) allows users to think in terms of logical datasets rather than understanding a distinction between physical tables and virtual views. The hierarchy of derived views provides a simple form of provenance; the user can inspect (and with permission, edit) the specific steps applied to produce the final result. The view-centric data model also affords collaboration: users can share the derived dataset (and its provenance) without emailing files that get out of sync with the master data. Moreover, collaborators can directly derive their own datasets in the same system, and the provenance relationship is maintained. The view-centric data model was a very successful feature in SQLShare: About 56% of the datasets in the system are derived from other datasets using views. Among the top 100 most active users, multi-layer view hierarchies were quite common. Figure 3.5 shows the max depth of dataset hierarchies forthese 100 users. A view that references only base datasets is assigned a depth of 0. Other users would use views as query templates: They would use apply the same query to multiple source datasets, copying and pasting the view definition and only changing the name of a table in the FROM clause.

Copy-and-paste seems inadequate here; motivated by this finding we intend to lift parameterized query macros into the interface as a convenience function[3].

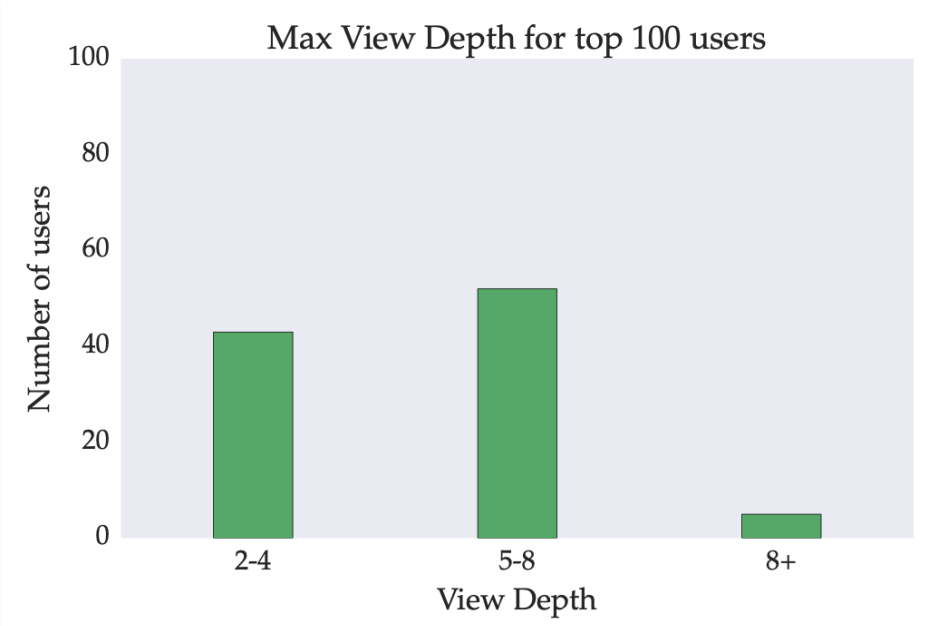


Figure 3.5: The maximum view depth for the 100 most active users of SQLShare. The data suggest that the ability to derive and save new datasets using views was an important features. More than 50% of users created chains of views deeper than five layers.

The view-centric data model also facilitates sharing: users can set dataset-level permissions, which are implemented in the database as view permissions. A dataset can either be private, public or shared with specific set of users. The permissions features were heavily used. About 37% of the datasets in SQLShare are publicly accessible, even though the default is to keep data private. About 9% of the datasets were shared with a specific other user. Moreover, about 2.5% of the views access other datasets that the author does not own, and over 10% of the queries logged in the system access datasets that the query author does not own. Beyond just collaborative analysis,

[3] A query macro would be different than a conventional parameterized query, since it allows parameters in the `FROM` clause rather than only as expressions.

the permissions feature allowed SQLShare to function as a data publishing platform. Several users cited SQLShare datasets in papers. One user minted DOIs for datasets in SQLShare; we are adding DOI minting into the interface as a feature in the next release.

3.3.3 Frequent SQL Idioms

SQLShare was designed to facilitate access to full, standards-compliant SQL as opposed to relying on the simplified SQL dialects often associated with analytics and sharing platforms (e.g., HIVE SQL [79], Google Fusion Tables [23]).

To evaluate whether full SQL was actually warranted, we counted the queries that use specific SQL language features that are sometimes omitted in simpler SQL dialects. As one might expect, queries involving sorting were comon (24%), and top k and outer join queries were frequent enough to justify support in any system (2% and 11% respectively). Perhaps more surprisingly, window functions (expressed using the SQL-standard OVER clause) appeared in about 4% of the workload. Virtually no systems outside of the major vendors support window functions; these newer systems will not be capable of handling the SQLShare workload!

In addition to specific SQL language features, we found evidence of recurring SQL "idioms" or "design patterns" that might motivate higher-level convenience functions to support query authoring. Aggregating timeseries and other data by computing a histogram was common enough (and awkward enough) that we are considering adding special support. Another common but tedious pattern was to rename a single column, and then be forced to explicitly list out every other column in the table. An expanded regular expression syntax raning over column names beyond just $\star$ is warranted: the ability to refer to all columns except a given column, or to replace a single column in its original order would be useful. More generally, the ability to refer to and transform a set of related columns in the same way would simplify query authoring: The expression `SELECT CAST(var* AS float) as $v FROM data` could indicate "replace each column with a prefix of `var` with an expression that casts it as a number and renames the expression appropriately." We also find frequent data cleaning and curation idioms in the SQLShare workload, we present a detailed analysis of these idioms in section 3.5.

3.4 Workload Analysis

In contrast to the conventional relational use cases characterized by a pre-engineered schema and predictable query patterns generated by the constraints of a client application, we hypothesized that SQLShare users would write queries that are more complex individually and more diverse as a set, making the corpus more useful for designing new systems. A more complex workload, especially one derived from hand-written queries, provides a more realistic basis for experiments in optimization, query debugging, and language features than a workload from a conventional, sanitized environment.

To test this hypothesis, we need to define metrics for query complexity and workload diversity. Since we are onboarding users with little or no database experience, query complexity needs to be measured in terms of the 'cognitive' effort it takes to express a task as a SQL query. Thus, the measures like query runtime or latency alone do not show the correct picture. In the discussion that follows, we have attempted to find proxy metrics to capture this 'cognitive' complexity. We develop simple metrics in this section and show that SQLShare queries on average tend to be more complex and more diverse than those of a conventional database workload generated from a comparable science domain: the Sloan Digital Sky Survey (SDSS) [40].

The SkyServer project of the SDSS is a redshift and infrared spectroscopy survey of galaxies, quasars, and stars. It led to the most detailed three-dimensional map of the universe ever created at the time. The survey consists of multiple data releases (10 to date), which represent different projects and different stages of processing refinement. Besides the survey data, the SDSS database contains one of the few publicly available query workloads from a live SQL database supporting both ad hoc hand-authored queries as well as queries generated from a point-and-click GUI. Singh et al., in the Traffic Report for SDSS [76] describe how during the first five years itself the system generated $180GB$ of logs. These logs were then normalized and cleaned and auxiliary data structures were built for analysis. SDSS is a useful comparison: it is a conventional database application with a pre-engineered schema but the users and tasks are not dissimilar to those of SQLShare.

3.4.1 SQLShare Queries are Complex

We interpret query complexity primarily as a measure of the cognitive load on the user during query authoring as opposed to computational compelxity in optimizing or evaluating the query. Our goal is to design lightweight data systems that can be used as part of day-to-day analytics tasks, which means we are competing directly with general purpose scripting languages for users' attention. Any query corpus that purports to reflect the usage patterns of analysts cannot rely on vanilla query patterns typically assumed in the database literature. In this section, we consider ASCII character length as a simple proxy for query complexity and then argue why the number of distinct operators is an improvement.

ASCII Query Length A naive estimate of query complexity from both the user and system perspective is the character length of the query as a string. The premise for character length as an indicator of complexity is is that the longer the query, the more a user has to write and read, and the more time and effort it takes to craft the query.

In both SQLShare and SDSS, most queries are short. But a significant number of queries in SQLShare, about 1500, are greater than 500 characters. This is not surprising given that users write queries over datasets which are often decomposed into multiple tables. Figure 3.6 shows the histogram of query length for both SQLShare and SDSS. SDSS has a high percentage of queries with similar length. We investigated this further and found that there are clear categories of SDSS queries corresponding to specific lengths. These categories correspond to particular query templates and example queries used many times, demonstrating that few of these queries should be considered hand-written. In addition, the shortest 20% of both workloads are less than 100 characters, which is quite short. But the longer queries in SQLShare range upto 11375 characters. However, query length does not necessarily capture cognitive complexity since long queries may involve repetitive patterns that are easy to write via copy-and-paste. We see examples of queries that are over 1000 characters long but involve just two operators (a filter applied to 50+ columns). The takeaway from the length comparison however is that SQLShare users write queries that the database community would consider unusual, which is precisely why this corpus is valuable. Such

queries should be considered in any realistic workload targeting weakly structured data and non-expert users.

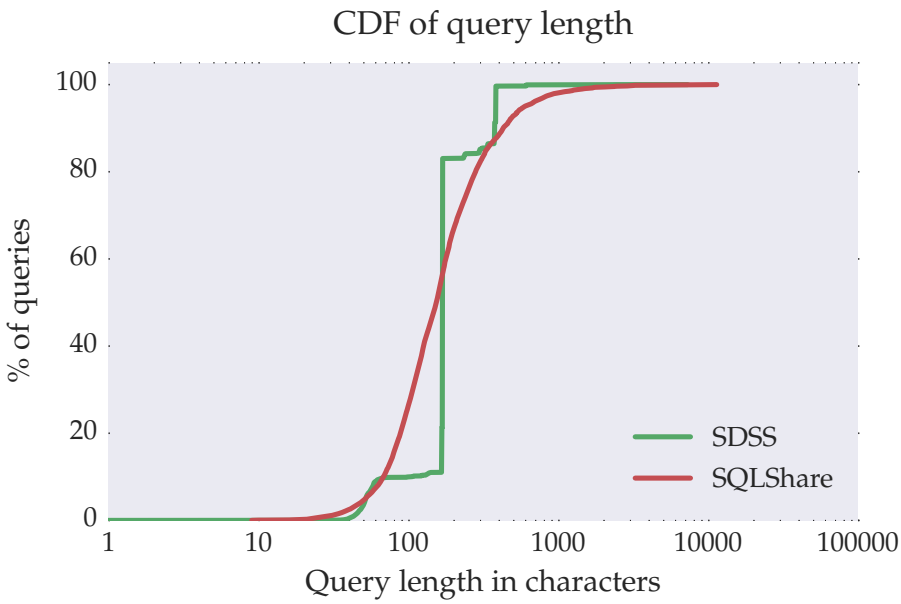


Figure 3.6: Hand-written SQLShare queries tend to vary more widely in length than SDSS queries. Short queries tend to be shorter, but long queries tend to be longer. SDSS queries show evidence of being "canned" rather than hand-written: only a few distinctive lengths are present and the majority of SDSS queries are about 200 characters. We explore these patterns in more detail in §3.4.2.

Distinct Operators in Query To capture the complexity of a query more accurately, we look at the number of operations in the execution plan. More operations mean more steps of computation which increases the complexity of scheduling of data flow for the system. SQLShare queries use a lot more operators than SDSS on an average. Many operations alone does not necessarily lead to a complex query for a user if the query uses the same operator over and over e.g. a union of 10 relations. A better metric to capture this case is to look at the *diversity* of operators and count the number of unique operators per query. A combination of both the number of operations and the

number of distinct operations intuitively captures complexity better than either of the two metric. Figure 3.7 shows the number of distinct operators for the three workloads.

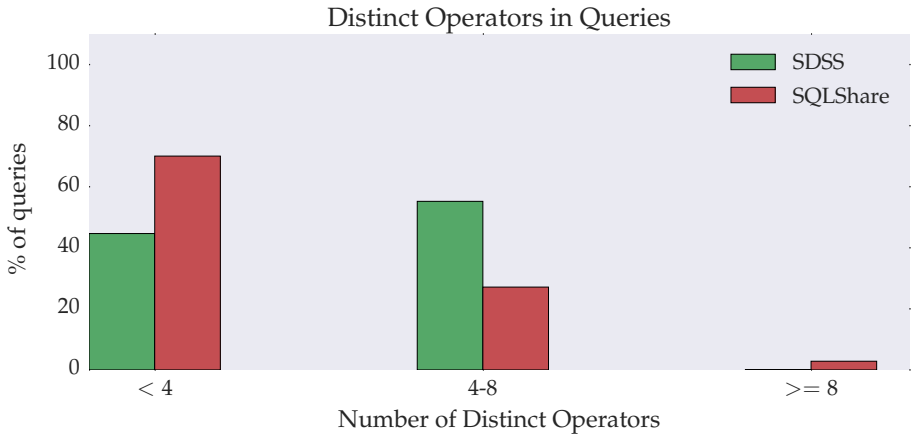


Figure 3.7: CDF of the number of distinct operators per query in both workloads. SQLShare has many queries with very few distinct operators, but about $5 - 8\%$ of the most complex queries have many more distinct operators than SDSS, suggesting that most complex queries in SQLShare appear to be more complex than the most complex queries in SDSS.

While a majority of queries in the SQLShare workload consist of < 4 distinct operators, a significant percentage of queries have significantly higher number of distinct operators, suggesting higher complexity. Among the top 10% of the queries with the highest number of distinct operators, the SQLShare queries tend to have almost double.

The next question one might ask is what type of the operators are present in the workload as a whole. This metric helps us understand workload complexity by providing the minimum requirement of SQL features for the workload to run, which is of interest to system designers.

Figure 3.8 and Figure 3.9 show the ten most common operators in SQLShare and SDSS. We ignored 'clustered index scan' for SQLShare workload because SQLShare uses SQLAzure as its backend and SQLAzure requires that every table be associated with a clustered index. SQLShare is dominated by aggregate queries, while SDSS has mostly computations on scalars most likely

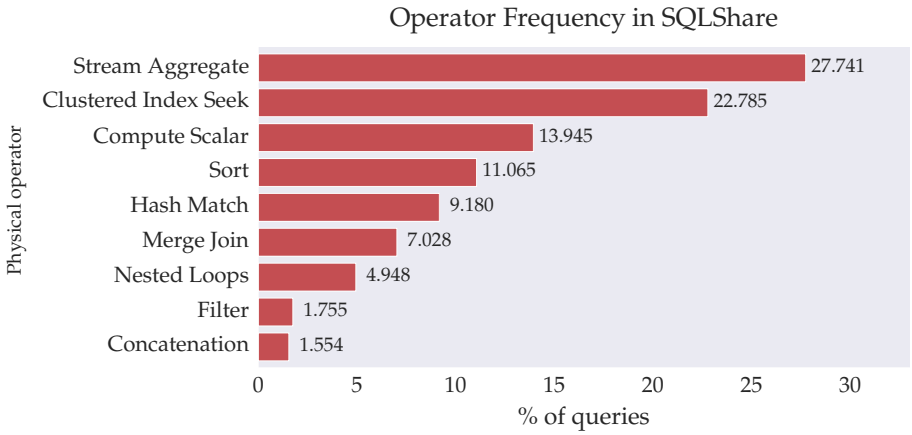


Figure 3.8: The most commonly used physical operators in SQLShare. We ignored Clustered Indexed Scan because SQLShare uses SQLAzure which requires them. Presence of a lot of aggregate and arithmetic operators in SQLShare suggests the presence of analytic workloads.

because it consists of a lot of user defined functions[4]. Overall, we find indications that users write more complex queries in SQLShare, suggesting that support for full SQL is useful for users.

3.4.2 SQLShare Queries are Diverse

If users are writing ad hoc queries rather than operating an application that generates queries on their behalf, we would expect that the *diversity* of the workload to increase. Rather than having the entire workload reduced to a few repeating templates, each query would be more likely to be unique. As the diversity of a query workload increases, system design and performance management becomes more challenging: up-front engineering and physical tuning becomes less efficacious. We consider high diversity a characteristic feature of the data science workloads and an important design goal of any system targeting this domain. This makes diversity an important feature to consider in database research. Current system overspecializes in simpler queries, since

[4] http://skyserver.sdss.org/dr5/sp/help/browser/shortdescr.asp?n=Functions&t=F

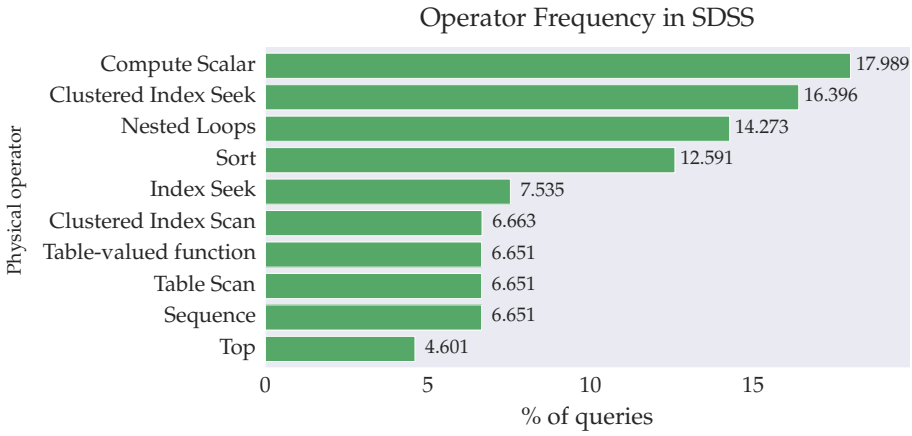


Figure 3.9: The most-used operator in SDSS is computation on scalars as a lot of queries use UDFs. Compared to SQLShare we see fewer arithmetic and aggregate operators.

a corpus of hand-written real queries is unavailable. For example, research on query recommendation platform like SnipSuggest [44] can be further improved by taking real science queries into consideration to reflect the full complexity of the problem. Similarly, query optimizers should consider optimizations for arithmetic optimizations as well. New languages or user interfaces which make common science idioms simpler would greatly increase scientists' productivity.

We can consider the question of whether SQLShare workloads are measurably more diverse than the workloads of a conventional database application such as SDSS.

Workload Entropy To quantify workload entropy, we must define query equivalence. A simple but naïve metric is exact ASCII string equivalence. ASCII string equivalence can only help eliminate very simple kinds of redundancies, such as identical queries generated by applications or repeated instances of copy-and-pasted sample queries. The SDSS workload contains both of these patterns, however, so we included this definition in our analysis.

A better measure of query equivalence was proposed by Mozafari et al. [67]: A query is represented by the set of all attributes (columns) referenced by the query. If two queries reference different sets of attributes, we say they are *column distinct*. A weakness of this metric in our context

is that the set of attributes referenced does not capture the user's intended task, and can therefore fail to distinguish queries that differ widely in layers of nesting, use of complex expressions (e.g., theta joins or window functions), and grouping structures. The presence or absence of these features may be what determines whether a query is perceived as "difficult" to novice users, which is an important consideration in the design of a system.

ASCII string equivalence overlooks equivalences and the column-based metric proposed by Mozafari et al. appears to overlook differences. We therefore propose a simple third metric by extracting a query plan and normalizing it by removing all constants. We obtain an optimized query plan from the database, which also contains estimated result sizes and estimated runtimes for each operator. The query plan resolves any heterogeneity resulting from the syntax (order of conditions, JOIN vs. WHERE, nesting vs. joins, etc.) In addition, we remove all constants and literals from the plan to create the *query plan template* (QPT). The QPT seems to offer a better description of the user's intended task, as it unifies most semantically equivalent queries but still incorporates the operations.

The SDSS workload initially contained $7M$ queries. However, after resolving redundant queries using simple string equivalence, the SDSS workload contained only about $200K$; 3% of the total. Many queries in the SDSS are actually not handwritten; they were generated by applications such as the Google Earth plugin or the query composer from the SkyServer website . In contrast, the SQLShare workload from 2011 to 2015 contains about $25K$ queries, 24096 (96%) of which were unique.

If we group queries by the set of columns referenced following Mozafari et al., we find that 45.35% of the queries are distinct in the SQLShare workload compared with only 0.2% of the 200K string distinct queries for SDSS.

Finally, SDSS only exhibits 686 unique query plan templates (0.3% of the $200K$ string distinct queries). The low entropy is not unexpected, given that many users manipulate a GUI and use standard examples queries to study a fixed schema. The SQLShare workload contains significantly higher entropy: It has about 15199 (63.07% of the 24096 string distinct queries) unique query plan templates. We summarize these findings in Table 3.3. While none of these measures are

Diversity Metric	SDSS	SQLShare
Total queries	$7M$	25052
String distinct queries	$200K$, 3% of $7M$	24096, 96% of 25052
Column distinct queries	467, 0.2% of $200K$	10928, 45.35% of 24096
Distinct query templates	686, 0.3% of $200K$	15199, 63.07% of 24096

Table 3.3: Workload Entropy: SQLShare queries are more diverse and have 63% distinct query templates. For SDSS, the number is very low (0.3%).

perfect, a high value of ($\frac{\text{Unique Queries}}{\text{Total Queries}}$) and a high absolute number of unique queries indicate high diversity in a workload, suggesting that the SQLShare workload is an appropriate test case to drive requirements of new systems.

We see two distinct usage patterns here in the two database-as-a-service platforms, SQLShare and SDSS. Since SDSS relies on a fixed, engineered schema, the diversity of queries they can ask is obviously limited. SQLShare allows users to upload arbitrary tables; there is no expectation that queries will overlap in form or content.

Expression Distribution Another measure of query diversity is the type and distribution of expression operators. We found the number of different expression operators to be 89 for SQLShare and 49 for SDSS. Moreover, we found that the workloads with intuitively higher variety not only use more diverse expressions but also more user defined functions (UDFs): SQLShare has 56 and SDSS 22. The most common intrinsic [5] and arithmetic expressions in SDSS are two scalar expressions followed by `BIT_AND`, `like` and `upper` (Table 3.4b). In SQLShare we found that six out of the ten most common expression operators (and again the vast majority) are operations on strings: `like`, `patindex`, `isnumeric`, `substring`, `charindex`, and `len` (Table 3.4a). Also, SQLShare has a higher expression diversity than other 2 workloads. This expression diversity backs our original hypothesis that use cases for SQLShare go beyond just data management

[5] https://technet.microsoft.com/en-us/library/ms191298(v=sql.105).aspx

Operator	Count
like	61755
ADD	31570
DIV	17198
SUB	13707
patindex	8212
substring	7490
isnumeric	7206
charindex	6364
MULT	4162
square	2636
len	2608

(a) SQLShare

Operator	Count
GetRangeThroughConvert	25746
GetRangeWithMismatchedTypes	25746
BIT_AND	21850
like	2376
upper	2312

(b) SDSS

Table 3.4: Most common intrinsic & arithmetic expression operators. String operations are very common on SQLShare, suggesting a lot of data integration and munging tasks.

and also include *data ingest, integration and cleaning* as suggested by the prevalence of string operations.

Reuse: Compress Runtimes The overall runtime of query is a measure of its complexity, but its misleading because runtime gets affected by the data size as well. However, runtime that can be saved by identifying re-occurring clauses in queries is a good measure of query diversity, i.e. lower reuse potential suggests higher diversity. Roy et al. show experiments in which 30% to 80% (depending on the workload) of the execution time can be saved by aggressively caching intermediate results [73]. Query optimization in the presence of cached results and materialized views is beyond the scope of this chapter. Nonetheless, we implemented a simple algorithm to

calculate reuse of query results that matches subtrees of query execution plans. While iterating over the queries, all subtrees are matched against all subtrees from previous queries. We allow a subtree that we match against to have less selective filters (filters are a subset) and more columns for the same tables (columns is a superset). If we find that we have seen the same subtree before, we add the cost of the subtree as estimated by the SQLServer optimizer to the saved runtime. Consequently, a precomputed intermediate result does not cost us anything when being reused.

Although this algorithm does not accurately model the actual execution time, we use it to estimate how diverse queries are. The algorithm can underestimate the potential for reuse since the matching misses cases when a rewriting would be needed. It could overestimate since we assume infinite memory as well as no cost for using a previously computed result. In this analysis we removed duplicate queries since a query that appears again will completely reuse previous results (recall that over 90% of SDSS queries are duplicates. But even for the distinct queries, 14% of the runtime could be saved. In SQLShare, we estimate saving to be around 37%. In all workloads, most of the saving per query was either very high (more than 90%) or very low (less than 10%). We conclude that most of the reuse could be achieved with a small cache if we have a good heuristic to determine which results will be reused. This also confirms our hypothesis that SQLShare queries are indeed more diverse.

3.4.3 Dataset Permanence Varies by User

In a conventional RDBMS, the schema is not expected to change much over time. We calculated the lifetime for the datasets in SQLShare, with lifetime defined as the difference in days between the first and the last time that dataset was accessed in a query and found that the SQLShare workload exhibits a variety of patterns. In particular, many users are operating in short-duration analysis loops, where they upload some data, write a few queries, and then move on to another task. This usage pattern is atypical for relational databases and seems to motivate new system features, including some of those already implemented in SQLShare.

Figure 3.10 shows the lifetime in days for the datasets for a user who both continuously updated new datasets and frequently accessed previous datasets. Each point represents one dataset for the

given user. The y-axis is the number of days between the first and last query that accessed the dataset. While many datasets are used across periods of years, the majority are uploaded once, analyzed over a period of 5-7 days, and never accessed again. For this user, over half are only accessed once, on the same day they are uploaded.

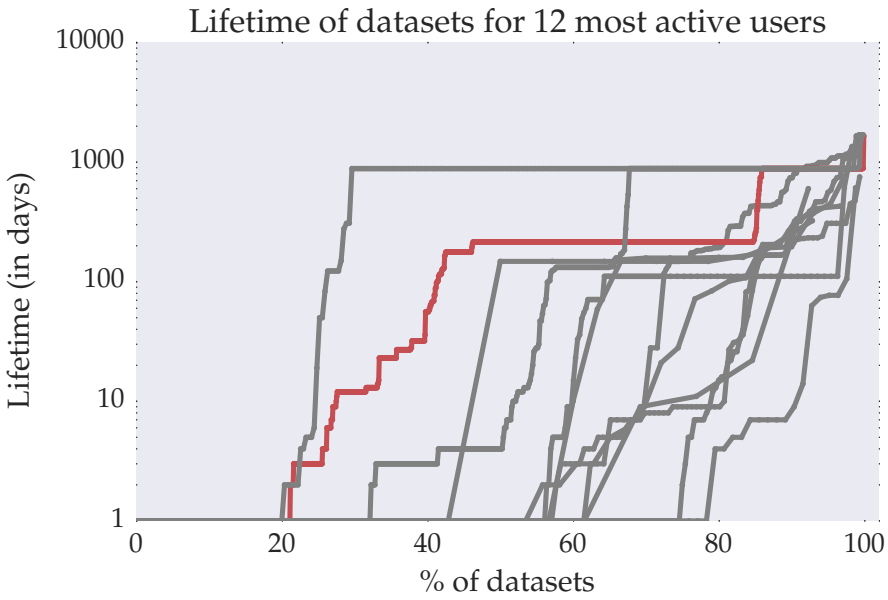


Figure 3.10: Dataset lifetimes for 12 most active users in SQLShare. Each curve is a user. The y-axis is the number of days between the first and last time the dataset was accessed. The x-axis is rank order. The great majority of datasets are accessed across a span of less 10 days, but some are accessed across periods of years. For some users, 80% of datasets have lifetime less than 1 day. This type of a workload, where a user explores the data and never accesses it again, is a departure from a conventional RDBMS use case. The highlighted user is the most active user in SQLShare.

Some users operate exclusively in a "data processing" mode, where they upload data at a predictable rate, process it using the same queries, and move on. The shorter dataset lifetimes associated with science and data science workloads is significant because it suggests that the costs associated with creating schemas and loading data would be incurred so frequently as to make these

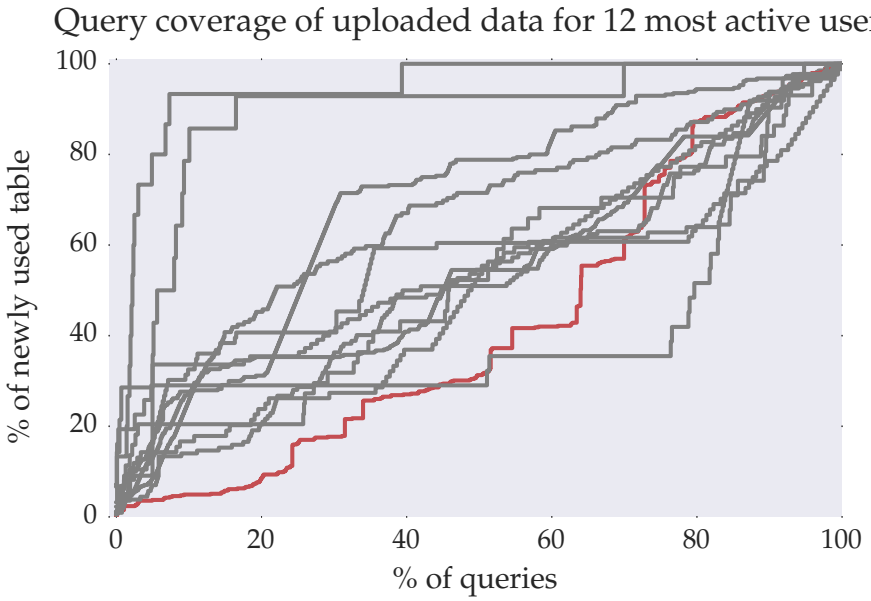


Figure 3.11: The rate of table coverage over time for the 12 most active users. Highlighted in red is the most active user for SQLShare. Each curve corresponds to a user and describes the percentage of tables accessed by the first N% of queries. A user who uploads one table at a time and queries it once would generate a line of slope one. Curves above slope one indicate a more conventional workload, where the user uploads many tables and then queries them repeatedly. Curves near to or below slope one indicate a more ad hoc workload, where queries are intermingled with new dataset uploads. We see both usage patterns in SQLShare, but the ad hoc pattern dominates.

workloads infeasible (or at least unattractive) for conventional database systems. This limitation does not exist in SQLShare due to the simple schema inference mechanisms and the web-hosted delivery vehicle.

SQLShare users in general are varied in their patterns of data upload and data analysis. Figure 3.11 shows table coverage for the most active users in SQLShare. Table coverage measures the cumulative count of tables referenced by queries upto a certain point in time. The figure shows how new datasets are being added all the time. This suggests that the use case is the following: user

keeps adding datasets, and the queries are usually written on *all* of the datasets taken together, and that she may be uploading data to overwrite/replace old tables then re-running the same queries.

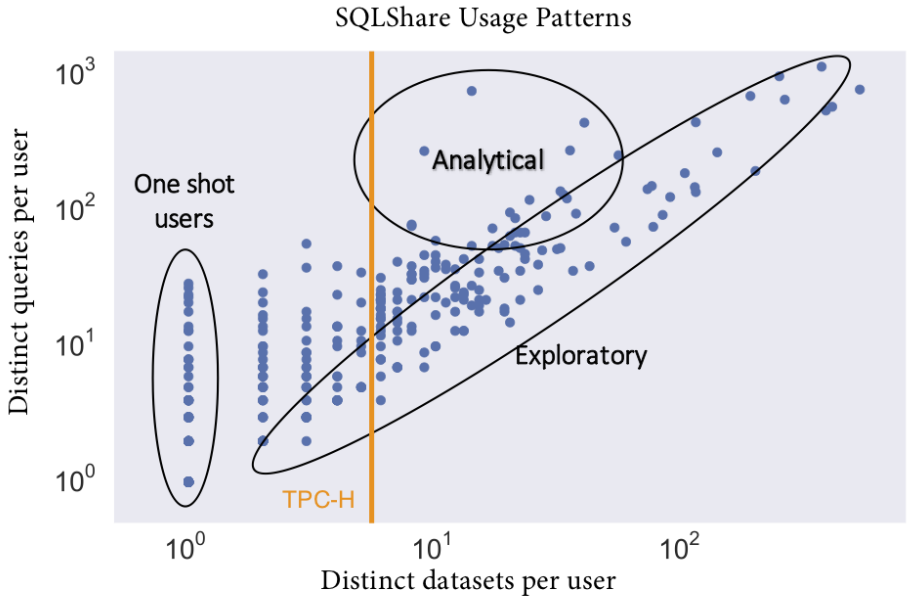


Figure 3.12: The ratio of datasets to queries suggests different usage patterns. Each point represents one user. The y-axis is the log of the number of queries of that user, and the x-axis is the log of the number of datasets of that user. Exploratory users only write a small number of queries over each dataset they upload. Some users exhibit a more conventional usage pattern, uploading a relatively small number of datasets and querying them repeatedly. A few non-active users upload one table and write very few queries. For comparison, standard benchmarks like TPC-H have a constant number of tables which are used in queries generated using a set of predefined query templates.

3.4.4 SQLShare Attracts High-Churn Work

Based on interviews with users, SQLShare seemed to be used to support workloads that exhibited higher "churn" than conventional databases, where datasets would be uploaded, queried a few times, and then put aside. To attempt to quantify our anecdotal evidence, we considered the ratio of

queries to datasets for each user, hypothesizing that this ratio would be very low for most SQLShare users.

Figure 3.12 shows the results. Each point is a single user. The x-axis is the number of distinct datasets owned by that user and the y-axis is the number of distinct queries submitted by that user. Both axes are on a log scale. The variety in workloads is also apparent: a few users upload relatively few tables (10-30) and query them repeatedly, which is reminiscent of a conventional database workload. These queries are labeled Analytical in the plot. But most users tend to upload approximately the same number of datasets as the number of queries they write, suggesting an ad hoc, exploratory workload. These users are labeled Exploratory. We also see that some users uploaded exactly one dataset, wrote 1-50 queries, and then never return. This group surely includes some users who were simply trying out the system and who either never had an intention to use it or did not find it useful. We label this group One-shot users.

To quantify the notion of workload diversity, we adapt the methodology of Mozafari et al [67]: break each user's workload into chronological blocks and measure the distance between the chunks. Each chunk is considered a separate workload and is represented by a row vector. Each position in this vector corresponds to a unique subset of attributes from the database. The value at that position represents the normalized frequency of queries that reference exactly this set of attributes. We then calculate the *euclidean* distance between these vectors, following Mozafari's algorithm [67]. The maximum distance found in the original paper was 0.003; this number was considered a high workload diversity. Among those users with sufficient queries to support this analysis, many exhibited *orders of magnitude* more diversity in their workload.

3.5 Data Cleaning in the Wild: Reusable Curation Idioms

Data curation is increasingly recognized as the bottleneck to analytics. Researchers and practitioners report spending a high proportion of their time cleaning, restructuring, transforming or otherwise preparing data for analysis. Worse, the time and effort spent on these "janitorial" tasks are difficult to amortize over repeated analysis projects; requirements tend to vary widely from project to project.

Classical approaches to data integration are relevant to curation, but tend to emphasize the design of a mediated schema to subsume two or more existing schemas. Data warehouse cubes are also associated with significant up front design and engineering of a centralized schema and the ETL workloads to fill it. These heavyweight "once and for all" approaches are a poor fit in data science contexts, where small teams of analysts convert data into actionable insights in more or less real time, drawing together multiple sources to answer targeted questions using specialized methods. Recent systems aim to reduce the effort required during the data curation step (e.g., format-busting and data profiling with Data Wrangler [2], enterprise integration with Tamr [77]), but scripts-and-files approaches are still dominant among data scientists: there is no time to amortize the up-front cost of warehouse design or global-as-view/local-as-view data integration exercises, and moreover, the data is rarely being extracted from a carefully engineered schema on which these methods tend to rely.

But the cost of this over-reliance on scripts and files is high: Our collaborators in the sciences report spending up to 90% of their time manipulating data [32], consistent with other anecdotal reports of the balance of time between data curation versus data analysis.

To reduce the burden of data janitorial work and improve reuse, we posit that databases can be naturally extended to support the entire data lifecycle, including preliminary data cleaning and curation from untrusted sources typically handled outside the database. That is, we argue that databases should be designed to *encourage* ingestion of dirty, weakly structured data (i.e., rows-and-columns but no engineered schema), and that curation should be performed directly in the database by writing SQL. In this section, we provide evidence of how this approach of letting databases do the curation via SQL queries, actually worked in practice by finding and characterizing *'cleaning idioms'* in a multi-year query workload.

We see multiple benefits to letting databases do this heavy lifting: i) the data always resides at one place during the entire analysis lifecycle (Figure 3.13), ii) the cleanup steps become more scalable, reliable, and reusable, and iii) the raw data is always directly available for reprocessing and recleaning in new contexts.

The primary disadvantage of this approach is the SQL authorship: many common curation

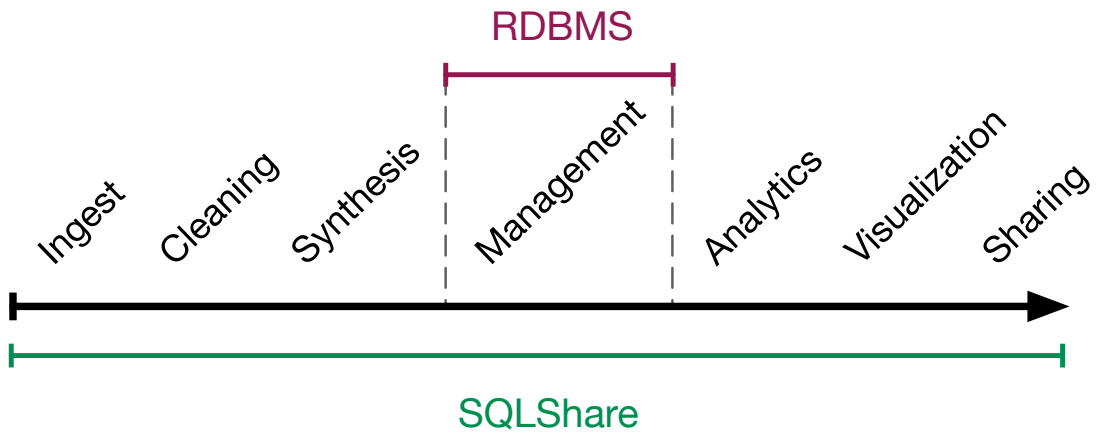


Figure 3.13: We find relational databases to be relevant at all stages in the scientific data lifecyle. SQLShare, a cloud-hosted database, empowers novice users by providing a system which handles use-cases across the data lifecyle.

tasks, while expressible in SQL, are sufficiently awkward as to prevent uptake. Our hypothesis was that direct support for a set of common SQL data curation idioms can make SQL-based curation competitive with script-based curation.

To understand data curation tasks in practice, we analyzed the workload of the SQLShare system [37, 36, 32], a Database-as-a-Service system targeting scientists and engineers. SQLShare encourages users to upload uncurated datasets over the web "as is," write queries across any datasets in the system, and share the results as views. The goal is to reduce the overhead in using relational databases in ad hoc analytics scenarios by reducing or eliminating upfront costs associated with installation, configuration, schema design, tuning, and ingestion. SQLShare supports automated schema inference and tolerates dirty data; these features allowed users to switch from managing and sharing brittle, dynamic sequences of scripts to a single system where all the operations can be performed safely, reliably, and scalably.

Over the years, we collected the query logs on SQLShare and analyzed interesting use cases. One common use case, as we expected, was that of data cleanup and curation tasks. We show in

this work how we can use these query logs to identify common cleanup tasks and provide them as suggestions for newer dataset uploads.

Furthermore, we envision how these cleanup *idioms* can be used to inform design of newer databases as follows:

1. Identify the clean up task from the query logs (hint: these are often the very first tasks performed on a dataset)

2. Generate templatized idioms for these cleanup tasks.

3. Upon newer dataset uploads, identify which idioms can be applied to the datasets.

4. Synthesize a clean up query from the selected idiom.

In this section, we identify common curation patterns that appeared prominently in the SQLShare workload, we describe how they are used in practice, and how these patterns informed specific features in SQLShare to assist in data ingest and query authoring. Finally, we describe some ongoing work in semi-automatic data curation based on these idioms.

3.5.1 Curating Idioms

The ubiquity of weakly structured data in the science use cases required SQLShare to tolerate (and even embrace) upload of weakly structured data. SQLShare encourages users to write SQL queries to repair and reorganize data rather than relying on offline scripts. By mining the workload, we extract generalizable patterns used to perform these repairs and use them to design services to partially automate cleaning tasks.

The SQLShare query corpus presents rich evidence to support this hypothesis. By searching the corpus of 4535 derived datasets (views), we found specific SQL idioms that correspond to schematization tasks: cleaning, typecasting, and integration.

Table 3.5: Frequency of observed idioms (total datasets: 4535)

Idiom	Datasets
Vertical recompositioning	100
Horizontal recompositioning	210
Column rename	720
NULL injection	420

We focus on the following curation patterns extracted from the SQLShare logs. Along with each idiom, we present an example query from the logs in Listing 2, and a method for using the idiom to support curation-on-ingest. Table 3.5 shows the frequency of occurrences of these idioms.

- *Vertical recompositioning*: Datasets in SQLShare are often representative of scientific processes where one logical dataset arrives in the form of several distinct files arriving at different times. For example, one lab collected data daily from a sensor depoyed in a local estuary. The need to pre-establish a schema and load the data file-by-file makes databases unattractive in these contexts, but SQLShare helped eliminate steps during data ingest. However, users still needed to craft a UNION ALL query to assemble the results, sometimes reordering columns or casting types to align the derived schemas.

  Curation on ingest: By learning these schema alignment heuristics automatically from the data, and applying schema matching methods, these UNION ALL queries can be automatically recommended and applied by the system upon data ingest. One such approach was describe in our previous work on automatically deriving example queries from base data [30].

- *Horizontal recompositioning*: This idiom pertains to horizontally partitioned datasets. As with vertical recompositioning, scientific processes generating the data sometimes produce

horizontally split data. Sometimes different labs working on same samples generate different attributes about them. These cases appear in the logs as multiway 1:1 joins.

Curation on ingest: Suggesting queries for horizontal recompositioning can be non-trivial. However, we can again use the approach shown in [30] to find potential for joins automatically. Automatic join finding using measures like jaccard similarity has been done in the past, combining this approach with a rich hand written query log to suggest data curation idioms is something that can finally make such approaches viable.

- *Column renaming*: It is common for datasets in SQLShare to have no column names in the source files. For this user scenario, SQLShare assigns default column names. Users are encouraged to write SQL to assign semantic names. We find evidence of 1996 uploaded tables, which is about 50% of the total tables, that had at least one default-assigned column name. The number of datasets for which all columns were assigned the default value is 1691. Almost 16% of datasets involve some kind of column renaming step, suggesting that users have adopted SQL as a tool for adding semantics to their data. We find this as sufficient evidence to back our hypothesis that the SQLShare workload contains a rich set of cleanup and curation queries.

  Curation on ingest: While identifying potential columns to rename is easy (columns with the default names are obvious candidates to begin, with a few false positives), suggesting valid renames is a very ambiguous problem. However, since we do have the advantage of having the previous tables and queries written on them. One approach could be to match the range of values of the column to rename to the range of values to previously existing and renamed columns. For example, for an attribute whose range is 0 to 360 and renamed to *'Angle'*, it might make sense to suggest for columns with values in the same domain. Another possible way could to be calculate the earth mover distance [74] between the histograms of column values and suggest rename to column with which this distance is least. There are other principled approaches as in WebTables [15] which uses the *attribute correlation statistics* to suggest schema auto-complete.

- *NULL injection and Type Coercion*: Sentinel values are routinely used to mark missing or inapplicable data; we see string values of "N/A" for example embedded in an otherwise numeric column. The SQL authors can use assemblies of CASE WHEN expressions, filtering, and type casting to replace these values with NULL or otherwise repair them. These constructs are conceptually trivial ("Across all columns, replace the value X with NULL") but hand-writing the corresponding query is tedious and error-prone. After removing bad tuples and replacing missing values with NULL, we find that about 200 of derived datasets used SQL CAST to introduce new types on existing columns.

  Curation on ingest: Our current implementation automatically infers data types based on a prefix of rows, and creates two table. The first table corresponds to the predicted type, and the second table holds non-conforming rows and has every column typed as a string. Finally, a view is created to union the 2 tables and is presented to the user, along with the information about the 2 base tables. This process helps separate the numeric data from the sentinel values, but does not automatically apply the CASE expressions.

Towards Idiom-Based Data Curation

So far we have shown the evidence of curation via SQL queries in the SQLShare workload. We discussed how these queries can be characterized into common curation idioms and finally we detailed the potential algorithms for curation on ingest.

Tying it all together, the idiom recommendation algorithm would work as follows:

- Identify the common curation idioms, the very first queries on a dataset are often representative of these idioms.

- Generate a query template for each idiom as shown in Qunits [68] and also in SQLShare analysis [36].

- At the time of data ingest, we use the **curation on ingest** techniques in section §3.5.1 to identify possible idioms.

- Synthesize the curation queries from the templates and provide them as suggestion to the users. The query synthesis problem has already been solved with multiple examples already available in the literature [13, 5, 8].

This approach of suggesting queries at ingest can save a lot of user time because writing these queries by hand can be repetitive and time consuming. The false positives don't hurt a lot because the user is always in loop and chooses which curation idiom, if any, she wants to apply to her dataset.

In our current implementation, we have a working analysis pipeline [36] and idiom detection. The next steps include integrating this pipeline with the SQLShare system and implementing a query synthesis algorithm.

```
Vertical recompositioning:
"SELECT * from [gbc3].[sqlshare-exp.txt]
UNION ALL
SELECT * from [gbc3].[gen_sqlshare.txt]"
Horizontal recompositioning:
"SELECT * FROM [che].[m1]
FULL OUTER JOIN [che].[m3]
ON
[che].[m1].m1_loci_id=[che].[m3].m3_loci_id"
Column rename:
"SELECT column2 as sp, column3 as SPID,
column4 as Prot FROM
[userX].[uniprotolyblastx2.tab]"
NULL injection:
"SELECT CASE WHEN [400 avg NSAF] = 0
THEN NULL
ELSE [2800 avg NSAF]/[400 avg NSAF] END
FROM
[emma].[NSAFwithAve]"
```

Listing 2: Example queries for each idiom.

Chapter 4

DATABASE AGNOSTIC WORKLOAD MANAGEMENT

In chapter 3 we presented a workload analysis for the SQLShare workload. Our hands-on experience with the workload analysis for SQLShare, along with a literature survey of the space of workload management (chapter 2) led us to realization that there is need for a workload analysis engine that operates independently of a database system.

Furthermore, in chapter 2 we noted how the features and the algorithms to extract them tend to be the significant contributions in the papers on workload management. But the state of the art in a variety of applications is to learn features automatically. For instance, Natural Language Processing applications previously relied on parsing and labeling sentences as a pre-processing step, but now use learned vector representations almost exclusively [22, 70]. This approach not only obviates manual feature engineering and pre-processing, but also has the potential to significantly outperform more specialized methods.

We see three trends motivating an analogous role for generalized workload representations. First, workload heterogeneity is increasing, making it difficult to maintain SQL parsers and feature extraction routines. The number of SQL-like languages is increasing, with inconsistent support and syntax for even relatively common features such as outer joins. Second, workload scale is increasing. Cloud-hosted, multi-tenant database services including Redshift [26], Snowflake [20], BigQuery [64] and more receive millions of queries daily from thousands of customers using hundreds of schemas; relying on brittle parsers (or worse, manual inspection) to identify query patterns that influence administration decisions is no longer tenable. Third, new use cases for centralized workload management are emerging. For example, SQL debugging [25], database forensics [69], and data use management [80] motivate a more automated analysis of user behavior patterns, and cloud-hosted multi-tenant systems motivate a more automated approach to query routing and

resource allocation.

In this thesis, we propose Querc, a database-agnostic system for mining and managing large-scale and heterogeneous workloads. We model workload management and analysis as a set of query labeling tasks. For instance, workload sampling can be reduced to labeling each query as present or absent in the sample, error prediction involves labeling each query with an error type, query routing involves labeling each query with a cluster resource to which the query should be routed, and so on. Because our framework depends only on the query text (along with typical metadata such as arrival timestamp and userid issuing the query), it can be used with any DBMS and any SQL dialect. In fact, as we will show, features learned with a workload against a particular schema and SQL dialect can be effective even when used with a *different* schema and SQL dialect.

The weakness of this approach is that it requires enormous amounts of data to be effective. But as database products migrate to the cloud, service providers have access to workloads from a large number of customers, potentially even across different database products. Since the input is just the query text, these diverse workloads can be processed as one very large dataset. But the resulting vectors can still be used to train models to support specific applications, as we will show on two representative tasks: workload summarization for index selection, user prediction for security audits and routing, and query error prediction.

4.1 System Architecture

Figure 4.1 illustrates the architecture of Querc. There are three applications, X, Y, Z. Each application has its own database, DB(X), DB(Y), and DB(Z), though these may be logical instances in the same physical multi-tenant service. In this example, DB(X) and DB(Y) are tenants in the same service. Each application is also associated with a separate stream of queries (at left), where query(X,t) indicates a batch of queries arriving for application X at time instant t.

Each application is associated with one Qworker, but each Qworker operates multiple classifiers. Qworkers may not be entirely stateless, as some labeling tasks process a small window of queries. However, the state is assumed to be small such that the Qworkers do not need their own local storage and can be load balanced and parallelized in typical ways. Each classifier is a pre-

trained (embedder, labeler) pair. The same trained embedder may be used across multiple applications. This split design is critical, because we want to learn features using a very large, combined workload, but an individual classifier may perform better when trained on an application-specific workload. In this example, application X and application Y both share the same embedder, EmbedderA, trained on the combined X and Y workloads, written EmbedderA(X,Y). This log sharing between customers may not always be permitted by customers for security reasons, and in this example, application Z uses only its own data. But there is some incentive for customers to pool their data as the additional signal can potentially improve accuracy, and some cloud providers support features to allow data sharing between customers.

The Labeler passes the query on to the database, but also transmits the query back to a central training module ("Training, Evaluation, and Offline Labeling" in Figure 4.1). The training module manages training sets, including the (parallel) execution of training and evaluation routines, then deploys trained models back to Qworkers. There is significant ongoing research in the database, systems, and ML communities on runtime architectures for training and deploying models (e.g., [57]); we do not discuss them further since our requirements are relatively modest.

Since Querc is specialized for query workload analytics rather than general machine learning, one data model can be shared among most applications. The only messages passed between components are labeled queries. A labeled query is a tuple $(Q, c_1, c_2, c_3, \dots)$ where c_i is a label. This simple model captures situations where a query arrives already equipped with a timestamp, a userid, an IP address, etc., but also captures more verbose query logs that are returned from the database.

The training module also records the queries with their predicted labels for retraining, evaluation, and to support offline analysis tasks. Offline tasks are those that do not require or do not allow processing each query separately, and can be implemented as typical batch jobs. For example, query clustering is important for workload summarization [49], but does not require real-time labeling of individual queries.

Training data is collected periodically from the databases in the form of query logs. These logs are (batched) sequences of labeled queries, but with additional labels to be used for training,

such as runtime, memory usage, error codes, security flags, resource IDs. We do not specify the mechanism by which these logs are transmitted from the database to Querc, since most systems have robust means of exporting logs in appropriate forms.

In some applications, Querc may not be in the critical path for query execution to avoid any performance overhead or reduce dependencies. In these cases, queries will be forked to Querc. No change to the architecture is required in this case; queries come in, and labeled queries are collected in the training module. The query is simply not forwarded to the database.

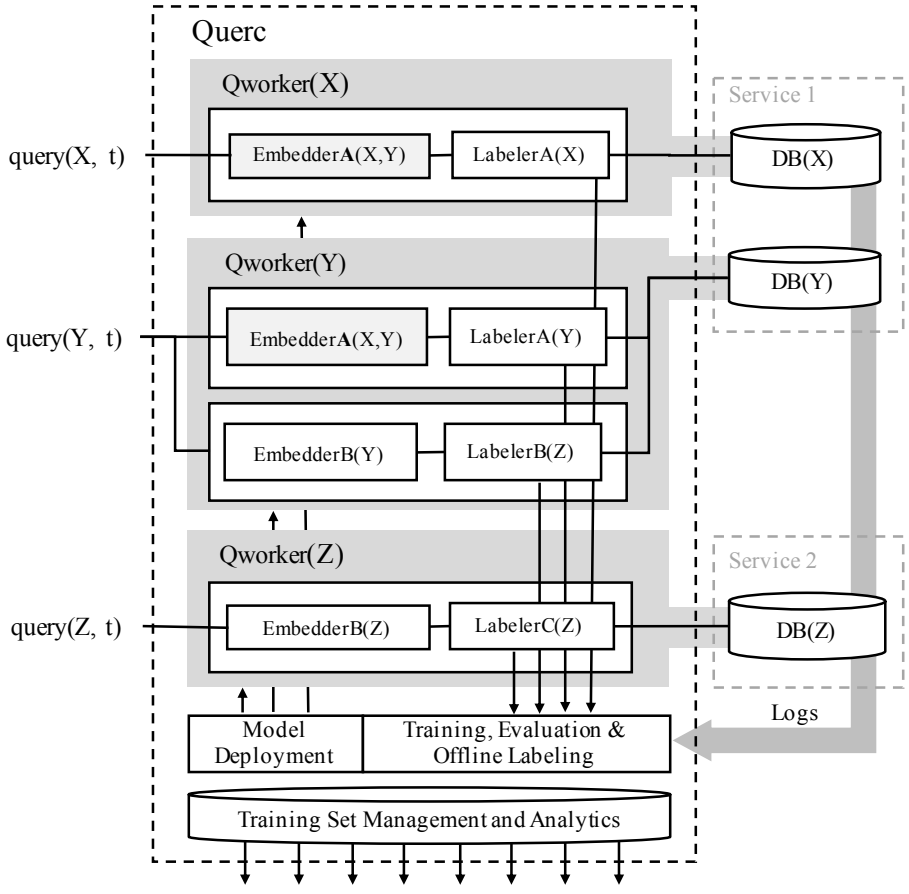


Figure 4.1: System architecture. Queries arrive for three different applications X, Y, and Z and are processed by one or more (embedder, labeler) pair before being sent on to the database, centralized for offline labeling tasks, or both.

This architecture is not designed for continuous learning, as the training is handled separately from real time query labeling. Not all algorithms can support fully continuous learning, and an important design goal is to support simple machine learning algorithms as labelers. Model training is therefore assumed to occur infrequently as a batch job.

4.2 Learning Vector Representations

There are multiple choices for embedders; we describe two initial models we evaluate in this chapter:

Context prediction models: Mikolov et al. [66, 65, 52] proposed learning a vector representation for words by predicting the next word in a context, and then deriving a vector representation for larger semantic units (sentences, paragraphs, documents) by adding a vector representing the paragraph to each context as an additional "word." The learned vector for this virtual context word is used as a representation for the entire paragraph. This "Doc2Vec" method has been shown to capture semantic relationships that work well for, say, sentiment classification and clustering tasks [47, 53]. This approach can be applied directly for learning representations of SQL queries: We can use fixed-size context windows to learn a representation for each token in the query, and include an identifier to learn a representation of entire query.

LSTM AutoEncoders: The paragraph vector approach in the previous section is viable, but it requires a hyper-parameter for the context size. There is no obvious way to determine a context size for queries, for two reasons: First, there may be semantic relationships between distant tokens in the query. Second, the length of queries vary widely in ad hoc workloads [36, 34]. To avoid setting a context size, we can use Long Short-Term Memory (LSTM) networks [86], which are modified Recurrent Neural Networks (RNN) that can automatically learn how much context to remember and how much of it to forget, thereby removing the dependency on a fixed context size. LSTMs have successfully been used in sentence classification, semantic similarity between sentences and sentiment analysis [78]. We use a standard LSTM encoder decoder network [87, 56] with architecture as illustrated in Figure 4.2.

An LSTM autoencoder is trained by sequentially feeding words from the query to the network

one word at a time, and then attempting to reproduce the input. The LSTM network not only learns the encoding for the samples, but also the relevant context window associated with the samples. The final output of the encoder network gives us an encoding for the query. Once this network has been trained, an embedded representation for a query can be computed by passing the query to the encoder network, completing a forward pass, and using the hidden state of the final encoder LSTM cell as the learned vector representation (of a predetermined dimensionality, for our experiments, we set this dimensionality to 300).

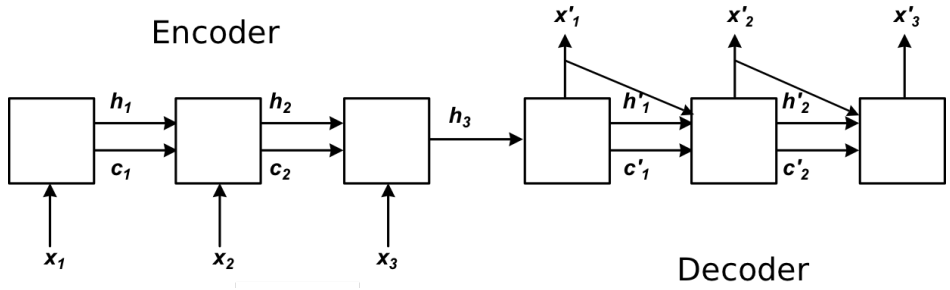


Figure 4.2: The LSTM Autoencoder network architecture learns to generate the input token in the decoding phase. The encoder component takes a token at a time as an input, maps these tokens to their corresponding vector representations and feeds the vectors to individual the LSTM cells. The vector representations for individual tokens, along with the weights of the network are learned using the standard back-propagation technique. Once trained, the encoder can be used to output a vector representation for the text of a query.

There are multiple prior approaches in the NLP literature that compare the efficacy of these models and their relative performance [52, 60, 78]. For this thesis, we consider context-based models (i.e., doc2vec) and LSTM AutoEncoders.

4.3 Applications

The applications supported by this system reduce to *query labeling*, and general workflow consists of two machine learning models: a representation learner (an embedder) and a classifier. We split

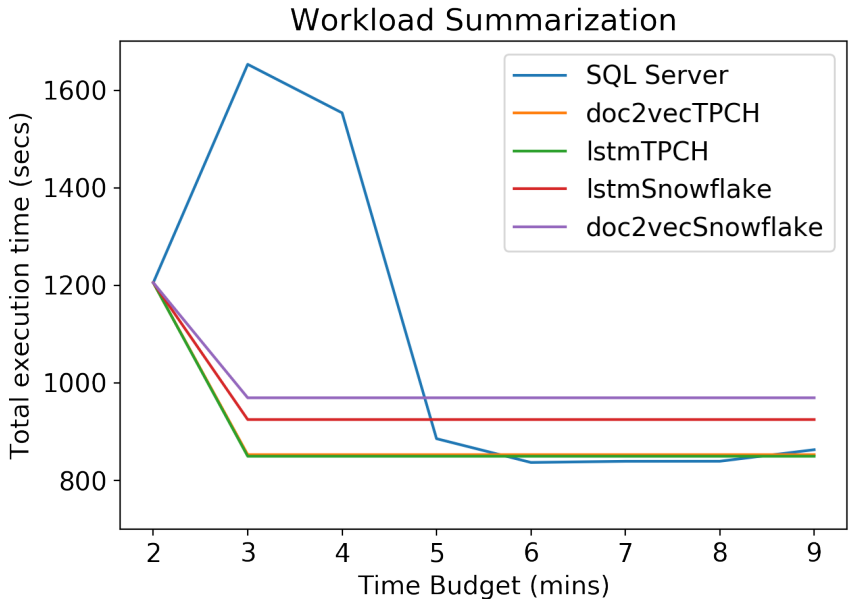


Figure 4.3: Workload runtime using indices recommended under various time budgets. For most time budgets, the workload summaries improve runtimes, even when the embedders were trained on an unrelated workload (lstmSnowflake and doc2vecSnowflake).

the task into two parts to allow the same representation to be used for multiple applications.

Workload summarization for index recommendation: The goal [17, 49] is to find a representative sample of the workload as input to further database administration, tuning, and testing tasks [17, 83]. In particular, workload summarization aids index recommendation, since the recommendation process is typically quadratic in the size of the workload [17]. While index recommendation systems are well-studied and ship with most production databases [17, 16], the quality of the representative sample determines the overall quality of the final recommendations. In Section 4.4, we show that a simple sampling procedure using learned features delivers a significant runtime improvement over the built-in sampling procedure in the SQL Server database system.

Enforcing query routing policies: Query Routing in a distributed database involves identifying the cluster resources on which to execute the incoming query. The policies that govern these

routing decisions may involve customer SLAs, security considerations (e.g., certain applications must use a physically distinct cluster from other applications), auditing requirements (e.g., queries from certain accounts or those accessing certain tables must be logged for auditing purposes). Even in modern cloud-hosted database products such as Snowflake [20] and BigQuery [64], these policies tend to be manually encoded, and management of these policies as they evolve, while maintaining multiple heterogeneous clusters used by thousands of customers, is increasingly perceived as untenable. Under the hypothesis that queries that follow a particular policy tend to have similar features, Querc can help identify policy misconfiguration by detecting when a predicted routing decision differs from the assigned routing decision.

Error prediction: Particular syntax patterns in the workload may be associated with resource errors or bugs in the database system. In a multi-tenant, multi-database, and high-volume scenario, identification of the syntactic patterns that tend to trigger errors, either manually or with scripts, becomes untenable: there may be hundreds of error codes, each with hundreds of subtle patterns that tend to trigger them, across hundreds of tenant schemas. Using learned features, a classifier to predict errors from syntax is trivial to engineer. This prediction allows the query to be routed to a different runtime environment that is instrumented, equipped with more more memory per node, or running a more stable version of the database engine.

In figure 4.4, we show a clustering of error-generating SQL queries from a large-scale cloud-hosted multi-tenant database system [20]. Color represents the type of error; there are over twenty different types of errors ranging from out-of-memory errors to hardware failures to query execution bugs (the figure highlights three specific error types). The syntax patterns in the workload are complex (as one would expect), but there are obvious clusters, some of which are strongly associated with specific error types. For example, the cluster at the upper right corresponds to errors raised when the compiler failed to parse a wrongly formatted DateTime field.

Using an interactive visualization based on these clusterings, the analyst can inspect syntactic clusters of queries to investigate problems rather than inspecting individual queries, for two benefits: First, the analyst can prioritize large clusters that indicate a common problem. Second, the analyst can quickly identify a number of related examples in order to confirm a diagnosis. For

Figure 4.4: A clustering of error-generating SQL queries from a large-scale cloud-hosted multi-tenant database system. Each point in the plot represents a query. The color represents the type of error (the figure annotates three specific error types). The syntax patterns in the workload are complex as one would expect, but there are obvious clusters some of which are strongly associated with specific error types. The legend lists the different error types that the query clusters correspond to. For example, the orange clusters correspond to queries that resulted in an error while parsing an incorrectly formated DateTime field and the cluster with queries annotated in green corresponds to queries that generated divide-by-zero error.

example, when we first showed this visualization to our colleagues at Snowflake [20], they were able to diagnose the problem associated with one of the clusters immediately.

Resource allocation: The structure of the query is not sufficient to accurately predict its runtime or memory footprint, but it can provide a hint that can be used for load balancing, scheduling, and as an input for optimization. If we can coarsely categorize queries as memory-intensive, long-running, etc. with some degree of accuracy, these labels can be used as a simple, database-agnostic way to speculatively allocate resources. Training data is readily available from the query logs themselves. We consider this application in a tech report companion to this thesis [35] and leave a detailed analysis for future work.

Query recommendation: The query recommendation problem can be modeled as a prediction of the next query the user will submit to the database based on the recent history of queries [7]. This prediction is then shown to the user though an appropriate client application to assist in query authoring. Our framework can generate features that can be used to train query recommendation models. We consider this application in a tech report companion to this thesis [35].

Security auditing: To the extent that users' individual workloads tend to follow predictable patterns, an anomalous query may be a sign that a user's account has been compromised. By formulating a prediction problem that tries to guess the user that submitted the query from the syntax alone, we can identify anomalous queries for security audits. In our framework, the labeler is a simple classifier $V \rightarrow user$.

4.4 Experiments

We consider two applications: Workload summarization for index selection, and labeling tasks for security audits and query routing.

4.4.1 Workload Summaries for Index Selection

The workload summarization task (with respect to index recommendation) is to find a subset Q_{sub} of a given query workload Q, such that the set of indices recommended based on Q_{sub} is similar to

the the set of indices recommended for the overall workload Q. Previous solutions are primarily variants of the approach of Chaudhuri et al. [17], which uses K-medioids to cluster the queries and selects a witness query from each cluster. However, the authors emphasize that a custom distance function should be developed for specific workloads; our hypothesis is that generic representation learning approaches obviate the need for these custom distance functions.

In the Querc framework, this task is offline and does not require real-time labeling of queries. Instead, we perform the task as an offline unsupervised learning task. In our approach, we assign each query to a vector (using a suitably trained embedder), then simply use K-means to find K query clusters and pick the nearest query to the centroid in each cluster as the representative subset. To determine K, we use an intentionally simple method (the "elbow method" [48]) which runs the K-means algorithm in a loop with increasing K till the rate of change of the sum of squared distances from centroids plateaus. Although better methods exist, we highlight the effect of the learned vectors rather than the choice of K. We present this workload summarization algorithm in Figure 4.5.

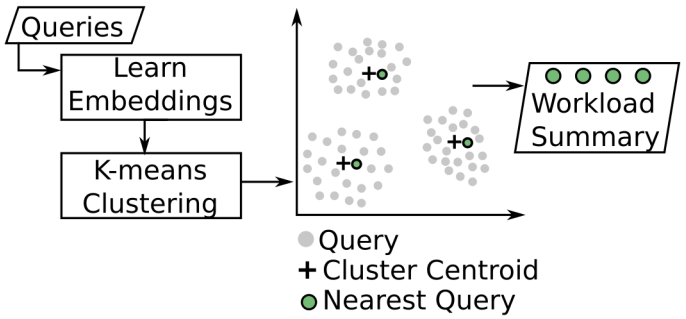
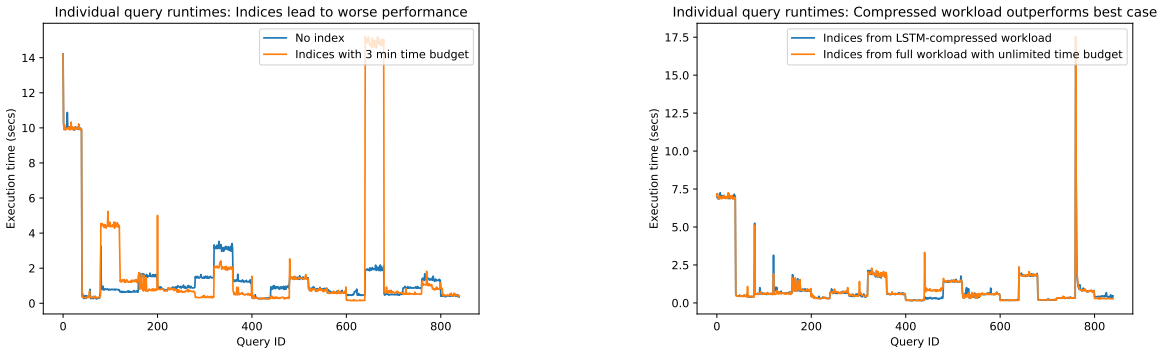


Figure 4.5: Workload summarization using learned query embeddings.

Setup: Following the evaluation strategy of Chaudhuri et al.[17], we first run the index selection tool on the entire workload Q, create the recommended indices, and measure the runtime t_{orig} for the original workload. We then run use the workload summarization algorithm to produce a reduced set of queries $\mathcal{Q}_{sub}$, re-run the index selection tool, create the recommended indices, and again measure the runtime t_{sub} of the entire original workload. We use SQL Server 2016 and the

Database Engine Tuning Advisor, which performs its own summarization on the input according to the documentation. We use an $m4.large$ AWS EC2 instance as the server. We use TPC-H with scale factor 1 as the workload for comparison with previous results and to interpret the recommended indices, but we also show how the method performs when trained on a more complex Snowflake workload.

We pass the summarized workload to the tuning advisor, along with a time budget (a parameter supported by the tuning advisor). Each experiment involves clearing caches, generating indices, applying the indices, and running the full workload. We report the time running the workload; the time budget specifies the time limit under which the advisor must return a set of recommendations.



(a) Runtime for each query under no indices and under indices recommended with a three-minute time budget. For a few specific queries (all instances of TPC-H Query 18), the presence of a recommended index results in significantly worse performance.

(b) Runtime for each query under our LSTM-based pre-compression scheme and the "optimal" indices recommended by SQL Server. The pre-compressed workload achieves essentially identical performance but with significantly smaller time budgets.

Figure 4.6: Comparing runtimes for all queries in the workload under different index recommendations.

Results: Figure 4.3 shows the results. The x-axis is the time budget, and the y-axis is the runtime for the entire workload after building the recommended indices. For time budgets less

	Account Labeling	User Labeling
Doc2Vec	78.8%	39%
LSTMAutoencodder	99.1%	55.4%

Table 4.1: Query Labeling results

than 3 minutes, the advisor does not produce any index recommendations for any method, and the runtime is constant at 1200 seconds. As we relax the time budget, different sets of indices are recommended, each associated with a separate runtime. The full workload (blue line) varies dramatically with the time budget, and surprisingly it gets worse before it gets better. For the summarized workloads, the workload is small enough that the runtimes are constant: Once three minutes have elapsed, the advisor has found the "optimal" set of indices, and allowing more time does not change the result.

We evaluate four trained embedders: two methods on two workloads. The two methods are Doc2Vec and the LSTMAutoencoder, and the two workloads are TPC-H itself, and a separate workload of 500,000 queries from the Snowflake service. When training the embedder on TPC-H (doc2VecTPCH and lstmTPCH), the advisor finds close-to-optimal indices in about three minutes as opposed to the six minutes the advisor requires on the full workload.

Surprisingly, under tight time budgets, the index recommendations made by the native system can actually *hurt* performance relative to having no indices at all! The optimizer chooses a bad plan based on the suboptimal indices. In Figure 4.6a and Figure 4.6b, we show the sequence of queries in the workload on the x-axis, and the runtime for each query on the y-axis. The indices suggested under a 3 minute time budget result in all instance of TPC-H query 18 (queries 640-680 in Figure 4.6a) taking much longer than they would take when run without these indices. The key conclusion is that *pre-compression can help achieve the best possible recommendations in significantly less time, even though compression is already being applied by the engine itself.*

Transfer Learning: Figure 4.3 also illustrates the capacity for transfer learning using Querc: When training the embedder on the snowflake dataset — a completely unrelated workload to TPC-H workload in the *SQL Server dialect* — the summarized workload still outperforms native SQL Server for most time budgets. This transfer learning effect allows us to bootstrap new applications without waiting for a representative workload to accumulate, and to avoid having to repeatedly re-implement brittle parsers and feature extractors for each new dialect of SQL we encounter.

4.4.2 Labeling for Security Audits

We consider the conditions under which the learned features from query syntax are sufficient to predict username and customer account, where each customer has many users. When the predicted username differs from the actual username, we can potentially flag the query for an audit. Predicting username can help flag queries for security audits, account and cluster labels can identify misrouted queries. labels from query syntax using the two embedding methods described in Section 4.2 over the Snowflake dataset.

Setup: We use embedders pre-trained on 500000 Snowflake queries. The experiment itself is run on another dataset of 200000 Snowflake queries labeled with username, account_id and cluster_name for the cluster that ran the query. Next we train classifiers (randomized decision trees) for username and customer account.

Results: Table 4.1 shows the results for the labeling experiments. The numbers denote the 10-fold cross validation score on the respective task. We find that LSTM based embedder beats Doc2Vec on all tasks. The LSTM method achieves near perfect accuracy when predicting the customer account, which is because it automatically incorporates signal from the schema, and different customers use primarily different schemas (there are instances of shared schemas, but that is the less common case). The method was completely generic and knows nothing about schemas or queries. For user prediction, the task is more difficult, and the overall accuracy is lower at 55%. Upon further analysis we found that the user labeling task has $> 95\%$ accuracies for a majority of accounts (Table 4.2). The accounts that had poor accuracies for user labeling had one distinctive property: multiple users running the exact same query, making the users nearly

#queries	#users	accuracy
73881	28	49.3%
55333	10	37.4%
18487	46	31.8%
5471	21	96.2%
4213	6	58.5%
3894	12	99.7%
3373	9	99.8%
2867	6	99.8%
1953	15	89.1%
1924	4	98.1%
1776	9	95.2%
1699	5	99.8%
1108	12	98.2%

Table 4.2: Top accounts with user prediction accuracy.

indistinguishable. In the sample of workload that we were working with, there were two accounts that had a number of repetitive queries by different users (for instance, 69% percent of the 74000 queries in an account had more than one user label), and these two accounts also covered around 65% of the total queries, bringing down the overall accuracy of classifiers.

4.4.3 Error Prediction

In this section, we perform a more quantitative experiment wherein we train a classifier to predict whether a query will raise an error or not. Such classifiers can be used in production to help quarantine error prone queries and run them on an instrumented debugging platforms.

Classifying multiple errors **Setup:** In this experiment we train a classifier which can predict an error type (or no_error) given an input Snowflake query. We use LSTMAutoencoder based embedder pre-trained on 500000 Snowflake queries. The classifier itself is trained using a dataset of 100000 queries from Snowflake. Next, we randomly split the learned query vectors (and corresponding error codes) into training (85%) and test (15%) sets. We use the training set to train a classifier. We present the performance of this classifier on the test set.

Results: We summarize the performance of the classifier on all error classes with more than 10 queries each in Table 4.3. The classifier performs well for the errors that occur sufficiently frequently, suggesting that the syntax alone can indicate queries that will generate errors. This mechanism can be used in an online fashion to route queries to specific resources with monitoring and debugging enabled to diagnose the problem. Offline, query error classification can be used for forensics; it is this use case that was our original motivation.

Although individual bugs are not difficult to diagnose, there is a long tail of relatively rare errors; manual inspection and diagnosis of these cases is prohibitively expensive. With automated classification, the patterns can be presented in bulk.

Classifying out-of-memory errors **Setup:** In this experiment, we compare the classification performance of our method for one type of error considered a high priority for our colleagues — queries running out of memory (OOM). We compare to a baseline heuristic method developed in collaboration with Snowflake based on their knowledge of problematic queries. We use a workload of 4491 Snowflake queries with a mix of queries with and without OOM errors to train a classifier to predict OOM errors. Following the methodology in the previous classification task, we use the pre-trained embedder to generate query representations for the workload, randomly split the learned query vectors into training (85%) and test (15%) sets, and present the performance on the test set.

Heuristic Baselines: We interviewed our collaborators at Snowflake and learned that the presence of window functions or joins between large tables in the queries tend to be associated with OOM errors. We implement four naïve baselines that looks for the presence of window functions

Error Code	Precision	Recall	f1-score	# queries
-1	0.986	0.992	0.989	7464
604	0.878	0.927	0.902	1106
606	0.929	0.578	0.712	45
608	0.996	0.993	0.995	3119
630	0.894	0.864	0.879	88
2031	0.765	0.667	0.712	39
90030	1.000	0.998	0.999	1529
100035	1.000	0.710	0.830	31
100037	1.000	0.417	0.588	12
100038	0.981	0.968	0.975	1191
100040	0.952	0.833	0.889	48
100046	1.000	0.923	0.960	13
100051	0.941	0.913	0.927	104
100069	0.857	0.500	0.632	12
100071	0.857	0.500	0.632	12
100078	1.000	0.974	0.987	77
100094	0.833	0.921	0.875	38
100097	0.923	0.667	0.774	18

Table 4.3: Performance of classifier trained using query embeddings for different error types (-1 signifies no error).

Method	Precision	Recall	f1-score
Contains heavy joins	0.729	0.115	0.198
Contains window funcs	0.762	0.377	0.504
Contains heavy joins OR window funcs	0.724	0.403	0.518
Contains heavy joins AND window funcs	0.931	0.162	0.162
LSTMAutoencoder	**0.983**	**0.977**	**0.980**
Doc2Vec	0.919	0.823	0.869

Table 4.4: Classifier performance for predicting OOM errors.

or a join between at least 3 of the top 1000 largest tables in Snowflake. The first baseline looks for the presence of heavy joins, the second baseline looks for window functions, and the third baseline looks for the presence of either one of the indicators: heavy joins **or** window functions, and the fourth baseline looks for the presence of both heavy joins **and** window functions. The baselines predicts that the query will run out of memory if the corresponding indicator is present in the query text.

Results: Table 4.4 shows the results. We find that our method significantly outperforms the baseline heuristics, without requiring any domain knowledge or custom feature extractors. We do find that the presence of heavy joins and window functions in the queries are good indicators of OOM errors (specially if they occur together) given the precision of these baselines, however, the low recall suggests that such hard-coded heuristics would miss a other causes of OOM errors. Querc obviates the need for such hard-coded heuristics. As with any errors, this mechanism can be used to route potentially problematic queries to clusters instrumented with debugging or monitoring harnesses, or potentially clusters with larger available main memories. We see Querc as a component of a comprehensive scheduling and workload management solution; these experiments show the potential of the approach.

Chapter 5

CASE STUDY: PREDICTING QUERY RUNTIME

Chapter 3 and 4 covered our solutions to the two challenges to the vision of DBMSs that learn from query workloads and runtime decisions based on changing workloads. We highlighted the lack of SQL query workloads, built a system that collects query workload and provided an analysis of the workload collected. Next, we demonstrated how techniques from NLP can benefit workload analytics. We also presented an architecture for Querc, a database agnostic workload management service.

In this chapter, we present a case study of using Querc and the ideas presented in the previous chapters to aid a very specific workload management task – query runtime prediction. While the problem setting and proposed solution in this chapter are specific to the Snowflake data warehouse service [20], they can be easily extended to other DBMSs. Accurate estimates of query runtimes (and other resources) are important for a variety of database management tasks, including but not limited to, query scheduling, appropriate selection of warehouse size (there by optimizing the monetary cost of query execution) and warning users about a potentially error prone heavy query (i.e. a query that should not have a long execution time in the absence of the bug) that can increase costs or hog the resources in a shared compute environment.

In particular, we are trying to answer the following research question:

- Can syntactic features of the query help the optimizer make more accurate runtime estimates?

We use Snowflake as a target system to implement and test our proposed ideas because of the availability to us of detailed historical query logs.

Problem Statement: Given a query Q, and an optimizer O, predict query runtime R and target cluster type (or cluster size or Degree of parallelism - DOP).

Similar to the experimental approach we took in chapter 4, we use learned representation of queries as syntactical features to train downstream runtime prediction model. We evaluate the benefit adding syntactic features features alongside those provided by the optimizer produced query execution plan.

The Snowflake query optimizer allocates resources (i.e. cluster size or DOP) to a query based on heuristics dependent on a variety of available information about the systems (e.g. I/O estimates, memory and cardinality estimates).

We notice that the choice the right cluster size (or DOP) is dependent on accurate estimations on query runtime (given a runtime estimate, an appropriate cluster size can be selected based on heuristics and SLA requirements). However, determining accurate query runtime (a regression task) is known to be a difficult problem and even the best optimizer estimates are often way off target [54, 28, 21, 27]. Therefore, we also evaluate a slightly simpler task of predicting whether or not a query is a long-running query (a classification task). Such a model can be used to aid query routing and scheduling.

5.1 Background & Challenges

Accurate estimation of query execution time is an open problem. The existing approaches either use optimizer estimates and cost models [28], or use machine learning on features extracted from query plans, while treating optimizer estimates as wrong or as a blackbox [27, 21].

We identify the following challenges to runtime estimation (or more broadly, resource prediction) for distributed cloud databases:

1. Lack of ground truth for training a resource prediction model:

    - Even in the presence of detailed query logs, the information available to us is the cluster type the query was run on, however, this decision could have been wrong to begin with, thereby making recorded runtime unreliable.

2. Selecting the right subset of the features available from the optimizer to augment the syntactic features.

3. Optimizer estimates and cost models are often inaccurate.

4. Lack of known evaluation metrics:

    - Runtime estimation is achieved using regression analysis. The usual metrics to evaluate performance of a regression model are root mean squared error (RMSE) and explained variance score. Both these measures are fairly opaque and do not help in determining the success of the downstream database tasks, e.g. query scheduling or predicting warehouse size.

In the next section, we present our approach that addresses these challenges for Snowflake. We then present some promising preliminary results and identify potential future directions.

5.2 Approach

Previous runtime estimation algorithms either treat the optimizer cost estimates as blackbox numbers [27, 21] or augment the existing cost models for better runtime estimation [28]. The features used for runtime estimation are extracted from query plans (e.g. memory estimates, number of joins, etc.) and the syntactic features in query workloads are largely ignored as the information pertaining to query runtime is assumed to be present in query plans. Past approaches fail to benefit from syntactic information (e.g. similarities between past queries that might indicate related runtimes). We therefore propose a hybrid approach that in addition to using various features extracted by the optimizer during physical plan generation, also uses syntactic features from queries. Using both syntactic and optimizer features makes our model more robust to potentially erroneous optimizer estimates or misleading syntactic patterns. As noted in the previous chapters, query embeddings can be used as syntactic features and are essentially "free" (i.e. require one-time offline training for embedding model).

The cluster type assigned to the query determines the query runtime as well. *True* cluster size and by extension *true* runtime are therefore not available in the training examples. One could potentially solve this by changing the prediction target to "total work done" by the system and work done could be approximated as a product of recorded runtime and DOP. The disadvantage of such an approach is that it assumes perfect scale up, which is rarely ever the case in real-life systems. We address the lack of ground truth by treating the cluster size or degree of parallelism (DOP, i.e. the number of threads allocated to the query by the optimizer) as a feature to the model. Our hypothesis is that given enough queries, we can learn the relationship between the query, DOP and the recorded runtime. More concretely, the training features for the model are query features (syntactic and from optimizer) and DOP, while the target variable is query runtime for the given DOP[1].

Another possibility we explored was to learn the relationship between allocated DOP and recorded runtime. More concretely, we collected a dataset of queries such that for each query, we allocate a different possible DOP and record corresponding runtimes. This *clean* dataset of (query, DOP, runtime) tuples is then used to train a regression model that predicts runtime given a query and DOP. However, this approach required rerunning customer queries for with varying DOPs. This meant that collecting a large enough dataset was prohibitively expensive (as Snowflake incurs cloud compute costs in rerunning each query) and therefore we leave a detailed analysis of this approach as potential future work.

We consider the following potential features that can be used to predict query runtime:

5.2.1 Features Sets:

1. Syntactic features:

    (a) Query2Vec(query_text): Query embeddings generated using the methods introduced in previous chapters.

[1]Such a model can then be used to predict different runtimes for a query, for varying DOP; and the DOP that minimizes the compute cost while meeting the SLA requirements can be assigned to the query.

2. Optimizer features:

   (a) DOP: the degree of parallelism or cluster size allocated to a query.

   (b) Top 10 scansets (based on the number rows in the scanset), where scanset is the I/O information about various scans present in the query plan. Each query plan has as many scansets as the number of data partition it touches.

   (c) Product of scansets: we multiply the number of rows in every scanset and use the result as a feature that acts as a proxy for worst case join cost.

   (d) Optimizer estimated memory cost for the query: This is a single number, optimizer estimate for the memory required by the query. We use this as a baseline feature because this number is used by the optimizer to guide the heuristics that currently determine DOP estimates for a query.

3. Metadata features:

   (a) Account ID: Categorical feature that identifies the customer account that was used to submit the query. Empirical observation is that certain account usually tend to submit more long running queries than short running queries. We use account id as a feature to verify if it can guide the model to make better decisions. Note that in table 4.1 we noted that query embeddings can help determine account id, so in some sense the embeddings already have account information. In this section we test if adding account id explicitly has an impact on runtime estimation.

   (b) Statement Type: Categorical feature that denotes SQL statement type (Select, Insert, DML etc.)

Additionally, we use the following as naive baseline features:

1. Query Length: Length of the query is the simplest possible syntactic feature. We use this baseline to validate if longer queries imply higher runtime.

5.2.2 Proposed evaluation metrics:

Regression models are often measured by RMSE and explained variance scores. However, both of these are difficult to interpret and do not actually help us determine the success of the original task (predict cluster type). Therefore, to get deeper insights into the relative performance of the models, we also look at rank-order correlation, i.e. we compute the correlation between the ranking of queries based on actual runtimes and the rankings based on predicted runtimes. Specifically, we calculate Spearman's rank correlation coefficient (Spearman's Rho) and Kendall rank correlation coefficient (Kendall's Tau) [39]. Both of these coefficients range from -1 to 1, with -1 indicating inverse correlation and 1 indicating perfect correlation.

Additionally, observing that the cluster sizes increase by a factor of 2 (at Snowflake), we define the following additional accuracy measure for our regression model:

Runtime Accuracy Score (RAS):

$$Score = \frac{\sum_{queries} \mathbb{1}(floor(log(AR)) <= log(PR) <= ceil(log(AR)))}{|queries|} \times 100$$

Where AR is the actual runtime for the query and PR is the predicted runtime by our model.

RAS tracks the percentage of predictions that are within the right logarithmic range of actual runtime. One possible approach to assign cluster size to a query is to take the log of its predicted runtime and use the corresponding value to guide the decision making rules that determine the cluster size (these rules would be system dependent, e.g. one such rule could be if $1 \leq log(PR) \leq 3 \rightarrow$ assign 'SMALL' cluster and so on).

The driving intuition behind this measures is that if the predicted runtime falls within a certain range of total runtime, the cluster size/DOP allocated would still be correct.

5.3 Experiments

We present some results for our approach in this section. We evaluate our methods on two tasks. First is a classification task to predict whether or not a query is a heavy hitter (i.e. whether or not query runtime is over a certain threshold). Second is a regression task that predicts actual query runtime.

We train models using different combinations of feature sets. All of these models are trained using 922164 random sampled queries from a day-long window of customer queries at Snowflake. The sampled queries originated from 803 separate customer accounts. The training was done on 80% of the data, the testing was done on the remaining 20%. Query embeddings were generated using a query2vec model pre-trained on 1000000 randomly selected queries.

5.3.1 Classification Task: Predicting heavy hitters

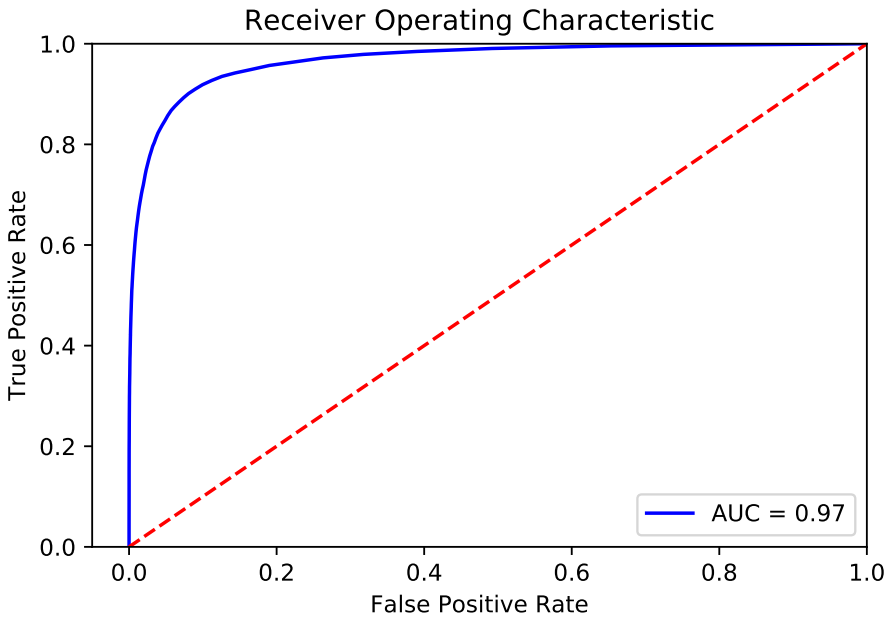


Figure 5.1: ROC curve for the classification task of predicting whether a query is a heavy hitter or not.

For this task, we classify queries into two categories, fast queries and heavy hitters (long running queries). Queries with runtime $>$ 30sec are labeled as heavy hitters. We report the performance of models trained using different feature sets in Table 5.1. Query length alone is a poor indicator of heavy hitters. Optimizer based features (e.g. memory estimates) perform fairly well with an overall f1-score greater than 90%. Using syntactic features alone results in a better overall

performance, but a poor recall for the heavy hitter class (implying a very high percentage of false negatives). Adding scanset information to the model gives the best performance improvement for predicting the heavy hitter class and boasts the recall. This is not surprising because scansets tracks the cardinality of base tables. Using metadata features helps improve the recall and f1-scores of the heavy hitter class as well. This is because adding a categorical feature that indicates account id helps the model track account that usually have long running queries. The performance of the model is lower for the heavy hitter class, one possible explanation for this is class imbalance as the number of long running queries are much lower (17756 heavy hitters versus 165510 short queries in our test dataset). Area under the Receiver Operating Characteristic (ROC) curve (AUC) is usually a good metric for evaluating performance models in presence of class imbalance. Figure 5.1 shows the ROC curve for the classification task using the model trained using all features (syntactic, optimizer based and metadata based). Our model achieves an AUC of nearly 100%, indicating a high confidence in the prediction made by the model.

5.3.2 Regression Task: Predicting exact runtime

Next we try different combinations of available features to measure the performance of regression models that predict query runtime. Table 5.2 summarizes the results. We report the RMSE, runtime accuracy scores and correlation coefficients for each of our model. The standard deviation for runtimes in our dataset was around 226 seconds, and all of our methods have RMSE lower than the standard deviation. We notice that the optimizer cost and syntactic feature (query embeddings) individually do not perform very well, however combining these features significantly improves the runtime accuracy scores and the correlation coefficients. Adding information from optimizer (scanset information) and metadata features improve the numbers even further. The best performance is achieved by using all features (syntactic, optimizer based and metadata based).

We compare our results against models trained on the baseline feature, query length. While the RMSE for the model trained using just the query length is lower than some other combinations of features, it is interesting to note that it does not necessarily translate to good performance for downstream tasks as the correlation coefficients and the runtime accuracy scores are much lower.

Classification Task							
	Class: Fast queries # instances: 165510			Class: Heavy hitters # instances: 17756			
Scores → Feature Set	Precision	Recall	f1-score	Precision	Recall	f1-score	Overall
Query length	0.96	0.77	0.86	0.25	0.70	0.37	0.77
Memory estimate	0.95	0.94	0.94	0.48	0.56	0.52	0.90
Query embeddings	0.92	0.99	0.96	0.91	0.22	0.35	0.92
Query embeddings + Memory estimate	0.92	0.99	0.96	0.91	0.22	0.35	0.92
Query embeddings + Scanset features + DOP	0.95	0.99	**0.97**	0.91	0.52	0.66	**0.95**
Query embeddings + Scanset features + DOP + Memory estimate + Metadata Features	0.96	0.99	**0.97**	0.91	0.57	**0.70**	**0.95**

Table 5.1: Predicting whether a query is a heavy hitter or not.

Given the results in table 5.1 and 5.2, we find sufficient evidence for the validity of our proposed hypothesis that syntactic features can aid runtime prediction and corresponding downstream tasks such as optimal cluster size selection.

5.4 Future Directions

We presented some preliminary results for the query runtime estimation task in this chapter. There are multiple extensions possible to this work.

Regression Task				
Scores → Feature Set	RMSE (seconds)	RAS	Spearman's rho	Kendall's Tau
Query Length	122.3	14.2	0.22	0.15
Memory estimate	128.1	16.6	0.25	0.2
Query embeddings	137.2	18.3	0.22	0.16
Query embeddings + Memory estimate	133.4	20.3	0.30	0.22
Query embeddings + Scanset features + DOP	**107.3**	30.2	**0.39**	**0.30**
Query embeddings + Scanset features + DOP + Memory estimate + Metadata Features	124.4	**33.9**	**0.39**	**0.30**

Table 5.2: Predicting query runtime using different combination of features, adding syntactic features help improve the regression performance.

Smarter features: Our work so far only used fairly simple features (i.e. information available in scansets). One potential extension is to try smarter higher order features like number of joins in the query, operator level cardinality and memory estimates and so on. In addition to these features, we can also imagine using text embeddings of templatized query plans that can capture information like plan shape and operator ordering.

Deploy and Test in production: The current experiments have been stand alone and have only measured simple statistics like RMSE and RAS, however, the real test of these methods would be actually trying them out in production. Our collaborators at Snowflake are building a

test pipeline to select DOP based on runtime estimate using the methods outlined in this chapter. For a given batch of queries, we will measure total time taken by all queries and average query time, for DOP selection based on our method and comparing that to the optimizer selected DOP.

Interpretability: Unlike optimizer estimates, which are driven by known heuristics and therefore explainable, our model is not easily interpretable. It is difficult to sell enterprise products that make seemingly non-deterministic choices to customers. One necessary extension to our work is to explore methods that can make the decisions made by our model more interpretable.

Estimating memory and other resources: Another potential extension to our work is to predict other resources like memory and I/O per query. We could also predict resources at an operator level granularity instead of a query level granularity. Predicting resource consumption per operator can potentially aid the optimizer cost models.

Chapter 6

CONCLUSIONS

Database management systems should learn from and model their behavior based on changing query workloads. However, current systems largely ignore the wealth of information the syntactic patterns in a workload provides. In this thesis, we lay down groundwork to build such systems by addressing two problems: first, a lack of publicly available SQL query workloads to inform database research in workload analytics and second, a lack of generic and robust workload analytics techniques that can augment existing databases with the ability to self-tune their performance.

We presented a new public query workload corpus for use by the database research community and described the open source SQL-as-a-Service system SQLShare that was deployed to collect it. Further, we showed that the features of SQLShare were instrumental in attracting new kinds of ad hoc queries that are written to perform tasks usually reserved for scripts and files. We performed an analysis of the SQLShare query corpus and argued that it is demonstrably more diverse than a comparable public workload in science, and that users are writing very complex queries by hand.

Next, we presented the architecture for Querc, a database-agnostic workload analytics service that captures the structural and schema patterns present in the query workload automatically, largely eliminating the need for the specialized syntactic feature engineering that has motivated a number of papers in the literature. The proposed architecture provides a new way of organizing a variety of database administration and user productivity tasks, and provides a mechanism by which to automatically adapt database operations to specific query workloads. Our evaluation of this architecture showed that our general framework outperformed or was competitive with previous approaches that required specialized feature engineering, and also admitted simpler classification algorithms because the inputs are numeric vectors with well-behaved algebraic properties rather than result of arbitrary user-defined functions for which few properties can be assumed. The use

of transfer learning in Querc allows workload analytics to be *SQL dialect independent* and enables the capability to bootstrap new analytics tasks and avoid re-implementing brittle code paths.

Moving forward, we will explore newer applications that Querc enables for database-as-a-service platforms like Snowflake [20]. One application to explore is query placement for a given batch of queries (e.g. queries in a stored procedure). At present, distributed databases execute all queries in a stored procedure on the same cluster, resulting in a potential waste of resources. Another potential application is combining error prediction and resource prediction at a per-operator level to detect and avoid memory overflows for instances such as large joins.

The results in chapter 4 demonstrate that the proposed framework in this thesis has the potential to use pre-trained models on generic workloads to aid analytics for previously unseen query. In future work, we will build this framework as a service which is accessible by third parties. Given the workloads that we have access to from Snowflake [20], such a service could be really beneficial for researchers who do not have access to massive query workloads.

In this thesis we have used techniques that have previously been used for learning representations for natural languages. The intuition behind such an approach was the following: since it is possible to learn representations for natural language (where the grammar is unknown), it should be possible (and even easier) to learn representations for structured code like SQL where the grammar is known and has much fewer rules than natural languages. However, an interesting future direction is to adapt representation learning methods specifically for languages like SQL. Learning methods tailored for SQL would also need to account for some of the differences in SQL and natural languages e.g. the difference in average length of SQL queries versus that of natural language sentences. The SQL queries in real-life workloads can span hundreds of thousands of tokens, whereas the methods used in this thesis are optimized for languages in which the average length of sentences is only in the tens of tokens.

In chapter 4 we showed how one can use NLP methods to learn representation for queries, another future direction is to explore the possibilities of generating queries given a dense representation. Encoder-decoder networks used in machine translation already achieves this for natural languages, and more recently Bunel et al. [14] showed how machine translation can be used for

program synthesis. The key insight is to predict the next syntactically token, instead of maximizing the likelihood of the next token from all possible words in the vocabulary, and essentially hand holding the model to predict only grammatically (i.e. syntactically) correct statements.

Generating queries from vectors would open up interesting new applications like workload generation. For example, one could start for a set of queries, learn dense representations for these queries and then generate more such vectors by adding noise in different dimensions. These new vectors could then be passed onto decoders as input and used to generate randomly generated, but syntactically correct queries. As noted in chapter 1 and 3, SQL workloads are hard to come by, therefore a workload generation scheme as described above would be of great value for database researchers.

BIBLIOGRAPHY

[1] Apache hadoop. https://hadoop.apache.org/. Accessed: 2014-10-14.

[2] Data wrangler. http://vis.stanford.edu/wrangler/.

[3] Sloan digital sky survey SkyServer. http://cas.sdss.org/. Accessed: 2014-02-12.

[4] TPC Benchmark H (Decision Support) Standard Specification Revision 2.14.0. http://www.tpc.org.

[5] Shadi Abdul Khalek and Sarfraz Khurshid. Automated sql query generation for systematic testing of database engines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 329–332, New York, NY, USA, 2010. ACM.

[6] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.

[7] Javad Akbarnejad, Gloria Chatzopoulou, Magdalini Eirinaki, Suju Koshy, Sarika Mittal, Duc On, Neoklis Polyzotis, and Jothi S Vindhiya Varman. Sql querie recommendations. *Proceedings of the VLDB Endowment*, 3(1-2):1597–1600, 2010.

[8] Javad Akbarnejad, Gloria Chatzopoulou, Magdalini Eirinaki, Suju Koshy, Sarika Mittal, Duc On, Neoklis Polyzotis, and Jothi S Vindhiya Varman. Sql querie recommendations. *Proceedings of the VLDB Endowment*, 3(1-2):1597–1600, 2010.

[9] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*, pages 390–401. IEEE, 2012.

[10] Sara Alspaugh. *Understanding Data Analysis Activity via Log Analysis*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2017.

[11] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[12] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[13] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *Knowledge and Data Engineering, IEEE Transactions on*, 18(12):1721–1725, 2006.

[14] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[15] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment*, 1(1):538–549, 2008.

[16] Surajit Chaudhuri, Prasanna Ganesan, and Vivek Narasayya. Primitives for workload summarization and implications for sql. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 730–741. VLDB Endowment, 2003.

[17] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 488–499. ACM, 2002.

[18] James Clark, Steve DeRose, et al. Xml path language (xpath). *W3C recommendation*, 16, 1999.

[19] Transaction Processing Performance Council. TPC-H benchmark specification. http://www.tpc.org/tpch/, 2008.

[20] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 215–226, New York, NY, USA, 2016. ACM.

[21] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603, March 2009.

[22] Yoav Goldberg and Omer Levy. word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[23] Hector Gonzalez, Alon Y Halevy, Christian S Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, Warren Shen, and Jonathan Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1061–1066. ACM, 2010.

[24] Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic, Fabrizio Silvestri, and Narayan Bhamidipati. Context-and content-aware embeddings for query rewriting in sponsored search. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 383–392. ACM, 2015.

[25] Torsten Grust and Jan Rittinger. Observing sql queries in their natural habitat. *ACM Trans. Database Syst.*, 38(1):3:1–3:33, April 2013.

[26] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923. ACM, 2015.

[27] C. Gupta, A. Mehta, and U. Dayal. Pqr: Predicting query execution times for autonomous workload management. In *2008 International Conference on Autonomic Computing*, pages 13–22, June 2008.

[28] Hakan Hacigumus, Yun Chi, Wentao Wu, Shenghuo Zhu, Junichi Tatemura, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1081–1092, Washington, DC, USA, 2013. IEEE Computer Society.

[29] Geoffrey E Hinton. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA, 1986.

[30] Bill Howe, Garret Cole, Nodira Khoussainova, and Leilani Battle. Automatic example queries for ad hoc databases. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1319–1322. ACM, 2011.

[31] Bill Howe, Garret Cole, Emad Souroush, Paraschos Koutris, Alicia Key, Nodira Khoussainova, and Leilani Battle. Database-as-a-service for long-tail science. In *Scientific and Statistical Database Management*, pages 480–489. Springer, 2011.

[32] Bill Howe, Francois Ribalet, Daniel Halperin, Sagar Chitnis, and E Virginia Armbrust. Sqlshare: Scientific workflow via relational view sharing. *Computing in Science & Engineering, Special Issue on Science Data Management*, 15(2), 2013.

[33] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model.

[34] Shrainik Jain and Bill Howe. SQLShare Data Release. https://uwescience.github.io/sqlshare//data_release.html, 2016. [Online;].

[35] Shrainik Jain and Bill Howe. Query2vec: NLP meets databases for generalized workload analytics. *CoRR*, abs/1801.05613, 2018.

[36] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. Sqlshare: Results from a multi-year sql-as-a-service experiment. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 281–293. ACM, 2016.

[37] Shrainik Jain, Dominik Moritz, and Bill Howe. High variety cloud databases. In *Proceedings of the 2016 IEEE Cloud Data Management Workshop.*, 2016.

[38] Debabrata Kar, Suvasini Panigrahi, and Srikanth Sundararajan. Sqlidds: Sql injection detection using document similarity measure. *Journal of Computer Security*, 24(4):507–539, 2016.

[39] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[40] Stephen M Kent. Sloan digital sky survey. In *Science with Astronomical Near-Infrared Sky Surveys*, pages 27–30. Springer, 1994.

[41] Alireza Khoshkbarforoushha and Rajiv Ranjan. Resource and performance distribution prediction for large scale analytics queries. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 49–54, New York, NY, USA, 2016. ACM.

[42] Nodira Khoussainova, Magda Balazinska, Wolfgang Gatterbauer, YongChul Kwon, and Dan Suciu. A case for a collaborative query management system. *arXiv preprint arXiv:0909.1778*, 2009.

[43] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. Snipsuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33, 2010.

[44] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. Snipsuggest: Context-aware autocompletion for sql. *Proceedings of the VLDB Endowment*, 4(1):22–33, 2010.

[45] Mi-Yeon Kim and Dong Hoon Lee. Data-mining based sql injection attack detection using internal query trees. *Expert Systems with Applications*, 41(11):5416 – 5430, 2014.

[46] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[47] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[48] Trupti M Kodinariya and Prashant R Makwana. Review on determining number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.

[49] Piotr Kołaczkowski. Compressing very large database workloads for continuous online index selection. In *Database and Expert Systems Applications*, pages 791–799. Springer, 2008.

[50] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[51] Aniruddh Ladole and Mrs DA Phalke. Sql injection attack and user behavior detection by using query tree fisher score and svm classification. 2016.

[52] Quoc V Le and Tomas Mikolov. Distributed representations of sentences and documents.

[53] Yann LeCun. The mnist database of handwritten digits.

[54] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[55] Omer Levy, Yoav Goldberg, and Israel Ramat-Gan. Linguistic regularities in sparse and explicit word representations. In *CoNLL*, pages 171–180, 2014.

[56] Jiwei Li, Minh-Thang Luong, and Dan Jurafsky. A hierarchical neural autoencoder for paragraphs and documents. *CoRR*, abs/1506.01057, 2015.

[57] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, Berkeley, CA, USA, 2014. USENIX Association.

[58] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[59] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 631–645, New York, NY, USA, 2018. ACM.

[60] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pages 142–150. Association for Computational Linguistics, 2011.

[61] Patrick Marcel and Elsa Negre. A survey of query recommendation techniques for data warehouse exploration.

[62] Ryan Marcus and Olga Papaemmanouil. Wisedb: a learning-based workload management advisor for cloud databases. *Proceedings of the VLDB Endowment*, 9(10):780–791, 2016.

[63] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212*, 2018.

[64] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[65] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[66] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[67] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. Cliffguard: A principled framework for finding robust database designs. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1167–1182. ACM, 2015.

[68] Arnab Nandi and HV Jagadish. Qunits: queried units in database search. *arXiv preprint arXiv:0909.1765*, 2009.

[69] Kyriacos E. Pavlou and Richard T. Snodgrass. Generalizing database forensics. *ACM Trans. Database Syst.*, 38(2):12:1–12:43, July 2013.

[70] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation.

[71] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment*, 6(10):853–864, 2013.

[72] MaryBeth Rosson and JohnM. Carroll. Active programming strategies in reuse. In OscarM. Nierstrasz, editor, *ECOOP 93 Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 4–20. Springer Berlin Heidelberg, 1993.

[73] Prasan Roy, Krithi Ramamritham, S Seshadri, Pradeep Shenoy, and S Sudarshan. Don't trash your intermediate results, cache'em. *arXiv preprint cs/0003005*, 2000.

[74] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. A metric for distributions with applications to image databases. In *Computer Vision, 1998. Sixth International Conference on*, pages 59–66. IEEE, 1998.

[75] Maja Rudolph, Francisco Ruiz, Stephan Mandt, and David Blei. Exponential family embeddings. In *Advances in Neural Information Processing Systems*, pages 478–486, 2016.

[76] Vik Singh, Jim Gray, Ani Thakar, Alexander S Szalay, Jordan Raddick, Bill Boroski, Svetlana Lebedeva, and Brian Yanny. Skyserver traffic report-the first five years. *arXiv preprint cs/0701173*, 2007.

[77] Michael Stonebraker, Daniel Bruckner, Ihab F Ilyas, George Beskales, Mitch Cherniack, Stanley B Zdonik, Alexander Pagan, and Shan Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[78] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification.

[79] 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. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[80] Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu. Automatic enforcement of data use policies with datalawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 213–225, New York, NY, USA, 2015. ACM.

[81] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A learning-based approach to the detection of sql attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–140. Springer, 2005.

[82] Yequan Wang, Minlie Huang, Li Zhao, et al. Attention-based lstm for aspect-level sentiment classification. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 606–615, 2016.

[83] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D. Viglas, and Allison Lee. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*, DBTest'18, pages 4:1–4:6, New York, NY, USA, 2018. ACM.

[84] Philip S. Yu, Ming-Syan Chen, Hans-Ulrich Heiss, and Sukho Lee. On workload characterization of relational database environments. *IEEE Trans. Softw. Eng.*, 18(4):347–355, April 1992.

[85] Hamed Zamani and W. Bruce Croft. Estimating embedding vectors for queries. In *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*, ICTIR '16, pages 123–132, New York, NY, USA, 2016. ACM.

[86] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[87] Richard S Zemel. Autoencoders, minimum description length and helmholtz free energy. NIPS, 1994.

[88] Mingyi Zhang, Patrick Martin, Wendy Powley, and Jianjun Chen. Workload management in database management systems: A taxonomy. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1386–1402, 2018.