SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment

Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, Ed Lazowska
Computer Science and Engineering Department, University of Washington, Seattle, WA, USA
{shrainik, domoritz, dhalperi, billhowe, lazowska}@cs.washington.edu

ABSTRACT

We analyze the workload from a multi-year deployment of a database-as-a-service platform targeting scientists and data scientists with minimal database experience. 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 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. We report on usage scenarios that suggest SQL is being used in place of scripts for one-off data analysis and ad hoc data sharing. The workload suggests that a new class of relational systems emphasizing short-term, ad hoc analytics over engineered schemas may improve uptake of database technology in data science contexts. Our contributions include a system design for delivering databases into these contexts, a description of a public research query workload dataset released to advance research in analytic data systems, and an initial analysis of the workload that provides evidence of new use cases under-supported in existing systems.

1. INTRODUCTION

We recently completed a four-year exercise to study the use of database technology in science and data science, where scripts and files are the de facto standard for data manipulation and analytics. Database technology has had limited uptake in these environments, in part due to the overhead in designing a schema and setting up a permanent database infrastructure. Changing data and changing requirements make it difficult to amortize these upfront costs, and as a result analysts tend to retreat to manipulating files with scripts in R, Python and MATLAB. Simultaneously, there is no appropriate corpus available for researchers to use to study this problem effectively. Industry partners do not retain or do not share query workloads, and certainly do not provide public access to data. Query workloads that do exist are associate with conventional pre-engineered database applications rather than oriented toward ad hoc analysis of ad hoc datasets.

Our hypothesis was that a relatively minor set of changes to how *existing* database technology is *packaged and delivered* would be sufficient to convince science and data science users to adopt SQL in their day-to-day data manipulation and analysis tasks, potentially displacing script-and-file based approaches. To test this hypothesis, we built SQLShare, a web-based system designed to make database technology easier to use for ad hoc tasks. We then deployed the system as a service and logged a multi-year query log as a research corpus for further analysis. This paper represents the release of the corpus to the database community, a description of the system that collected it, and an initial analysis of the data it contains. We also performed some informal ad-hoc interviews with our active user base, which is mostly comprised of *researchers and scientists*, and some of the results and conclusions drawn are influenced by our familiarity with these users..

Our ultimate goal is to design new systems to better support ad hoc data management and analytics. Although we target researchers in the physical, life, and social sciences, we find that the requirements in data-intensive science and data-intensive business have begun to converge. The data scientists that are in high demand share characteristics with academic researchers: a stronger math background and deep domain knowledge, traits that sometimes come at the expense of software and systems expertise. In response, we designed SQLShare [15, 14] to reduce the use of a database to a minimal workflow of just uploading data, writing queries, and sharing the results — we attempt to automate or eliminate steps for installation, deployment, schema design, physical tuning, and data dissemination. We deployed this system to determine its efficacy in attracting science users and understand their requirements. This experiment was not a controlled user study; rather, we deployed SQLShare as a production service — an "instrument" to collect measurements on the kind of datasets used in practice, the kinds of queries these scientists and data scientists would write, and how the specific SQLShare features would be used.

In this paper, we describe the corpus we collected during this experiment, describe the SQLShare system used to collect the corpus, and present findings from an initial analysis of the corpus. Our results show that the specific features of SQLShare are associated with specific usage patterns observed in the workload dataset, and that these patterns may motivate new kinds of data systems. We have made the query log dataset available to the research community to inform research on database interfaces, new languages, workload optimization, query recommendation, domain-specific data systems, and visualization. Our dataset is one of the only two known real-world SQL workloads publicly available to the database

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2882903.2882957

research community (the other being the Sloan Digital Sky Survey workloads [18]), and the only one containing primarily ad hoc, hand-written queries over user-uploaded datasets.

The workload suggests that some of our users are comfortable expressing complex tasks in SQL (binning, integration across many tens of datasets) and making use of advanced language features such as window functions. Further, there is evidence that by relaxing schema requirements and inferring structure automatically, we allow analysts to use SQL itself to incrementally impose structure by writing views: assigning types, renaming columns, cleaning bad values, reorganizing tables — tasks normally associated with offline preprocessing in a general purpose language.

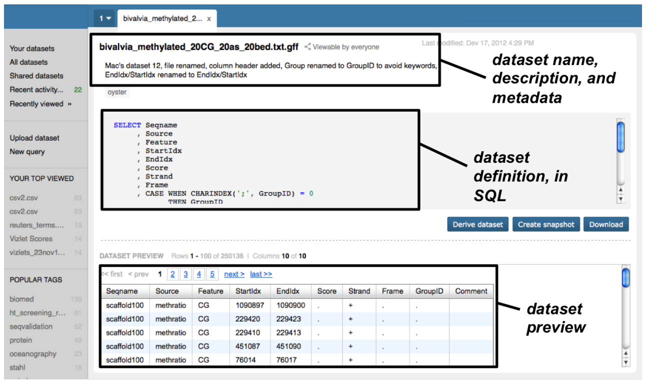
Although SQL itself appears useful in this context, support for complex query authoring emerges as an important research topic: query recommendation, editors, debuggers, new languages supporting the common idioms found in this workload all appear well-motivated. SQL has enjoyed a resurgence as an important interface for analytics without necessarily relying on engineered schemas (e.g., HIVE [31], SparkSQL [4], and Schema-free SQL [22]). More recently Metanautix [2] has adopted SQL for image and video processing tasks. Our work provides further evidence of SQL's utility in non-traditional contexts, especially in science.

We find that conventional database views are remarkably useful for a variety of tasks in science and analytics environments: protected data sharing, workflow provenance, data publishing, abstraction for data processing pipelines. To encourage the use of views, we made view creation a side effect of query authoring and were careful to erase any distinction between physical tables and virtual views — both views and tables were considered a "dataset."

Sharing results with collaborators without emailing scripts and files emerged as an important use case. Collaborators reported browsing long chains of nested views to understand the provenance of a dataset. Snippets from one query were reused in other queries routinely. Shared datasets could be queried and manipulated without requiring data to be downloaded first. Altogether, these features led to workloads that involved very short data lifetimes: Data could be brought into the database easily, analyzed briefly with SQL, and shared with other partners with a click. The users report that these short-lifetime workloads were typically implemented as scripts in R or Python; SQLShare demonstrates that they can be supported in the database itself with minimal system changes.

We intend the database community to use this paper as a reference for understanding how non-experts use SQL "in the trenches," as a description of a workload dataset to drive further research, and as a baseline for comparison with new database systems targeting similar users. This paper makes the following contributions:

- 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.
- A description of the important features of the SQLShare, a system designed to increase uptake of database technology for ad hoc analysis and deployed as the instrument used to collect the workload.
- 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.
- A comparison of the SQLShare workload with that of a conventional schema-first science database to provide evidence that the characteristics we uncover are unique to the SQLShare experiment.



bivalvia_methylated_20CG_20as_20bed.txt.gff Viewable by everyone Last modified: Dev 17, 2012 4:29 PM

Mac's dataset 12, file renamed, column header added, Group renamed to GroupID to avoid keywords, EndIdx/StartIdx renamed to EndIdx/StartIdx

oyster

```
SELECT Seqname
     , Source
     , Feature
     , StartIdx
     , EndIdx
     , Score
     , Strand
     , Frame
     , CASE WHEN CHARINDEX(';', GroupID) = 0
          THEN GroupID
```

DATASET PREVIEW Rows 1 - 100 of 250136 | Columns 10 of 10

<< first < prev 1 2 3 4 5 next > last >>

Seqname	Source	Feature	StartIdx	EndIdx	Score	Strand	Frame	GroupID	Comment
scaffold100	methratio	CG	1090897	1090900	.	+	.	.	
scaffold100	methratio	CG	229420	229423	.	+	.	.	
scaffold100	methratio	CG	229410	229413	.	+	.	.	
scaffold100	methratio	CG	451087	451090	.	+	.	.	
scaffold100	methratio	CG	76014	76017	.	+	.	.	

Figure 1: Screenshot of the SQLShare user interface. Each dataset is a view associated with some descriptive metadata and a preview. Creating and sharing views is the primary workflow in using the system.

2. PROBLEM CONTEXT

Relational databases remain underused in science (and data science) despite a natural fit between hypothesis testing and interactive query. Some ascribe this underuse to a mismatch between the requirements of scientific data analysis and the models and languages of relational database systems [29]. Our experience is that rows-and-columns datasets remain ubiquitous in science, and that standard relational models and languages remain a good conceptual fit. We find that the key barriers to adoption lie elsewhere:

- Although collections of records are generally appropriate, datasets in practice are *weakly structured*: they exhibit inconsistent types, missing and incorrect values, inconsistencies across column and table names, horizontal and vertical decomposition of logical datasets into sets of files. General purpose languages and collections of files are perceived to be the highest level of abstraction that can accommodate this heterogeneity.
- Daily activity results in long chains of *derived datasets*. Raw data are cleaned, restructured, integrated, processed, and analyzed — each step in the chain results in a derived dataset. The processing history of the dataset is important for provenance purposes. These derived datasets may be virtual or materialized, but in either case ease of recomputation is considered paramount. The natural inclination is to model the processing pipeline as a sequence of scripts, giving rise to workflow management systems designed to manage the execution and sharing of script-based processing [7, 9, 24, 30].
- As workloads become increasingly analytical, declarative languages are perceived to be increasingly limited. The emergence of impoverished SQL languages that lack support for standard features exacerbates the problem.[1]
- Inter-institution collaboration demands selective *sharing* of data, analysis steps, and results on-demand.
- Teams of researchers exhibit significant *diversity*, with varying backgrounds, varying experience levels, and varying tolerance for technical barriers. Data systems that require significant installation, configuration, and loading steps before

[1]For example, the popular HIVE SQL dialect only supports subqueries in the FROM clause and does not support window functions, complicating common analytical idioms that are naturally expressed in GPLs using loops.

Table 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
Diverse users	SaaS	Broad use
Low data lifetime	Relaxed schemas	"One-pass" workloads

delivering value are perceived as "maybe good for database experts, but not right for me."

- Highly dynamic analysis environments result in *short data lifetimes*: datasets are presented to the system, analyzed, and then put aside. In contrast, conventional database applications tend to emphasize a permament, pre-engineered schema. The transient nature of data makes it difficult to amortize the cost of schema design and data loading in a conventional database and motivate a tighter interaction loop for incorporating new data sources.

Given these requirements, there is a temptation to design a new system from scratch targeting these requirements. Instead, we consider the null hypothesis: that existing database systems are largely equipped to support these new environments, provided we change their interfaces to support these new workflows. We consider how to change the "delivery vector" of relational databases by emphasizing certain features (full SQL, views) and de-emphasizing or automating the use of other features (fixed, pre-engineered schemas).

In this paper, we first formalize these requirement and show how we address them in SQLShare. Next we analyze SQLShare workload to show how we fared on fulfilling these requirements over the last 4 years. We then look at a comparison of SQLShare workload with the SDSS workload and show how SQLShare queries are more diverse and very complex. We conclude with a discussion about the non-traditional workflows generated over SQLShare. We show that by making use of databases simpler, SQLShare enabled novice database users to build data analytic skills and focus on science and domain expertise.

3. SQLSHARE PLATFORM FEATURES

The SQLShare [14] 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. Table 1 summarizes the key features required to support science and data science use cases. Next we look at how each of these feature 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 the success of the experiment: The backend system was developed 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 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. Somewhat surprisingly, almost 50% of the datasets uploaded did not have column names supplied in the source file. Once a schema is derived, the

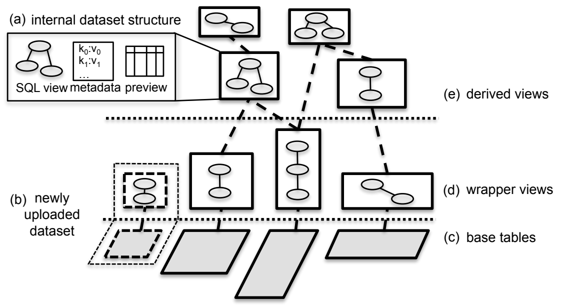


Figure 2: The SQLShare data model. (a) The internal structure of a dataset, consisting of a relational view, attached metadata, and a cached preview of the results. (b) A newly uploaded datasets creates both a physical base table and an initial (trivial) wrapper view. (c) The physical data are stored 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.

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.

3.2 Data Model: Unifying Views and Tables

We illustrate the data model of SQLShare in Figure 2.

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, 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 $V2 \to V1 \to T$ 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.

3.3 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.

3.4 Architecture

The architecture of SQLShare appears in Figure Figure 3. 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 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

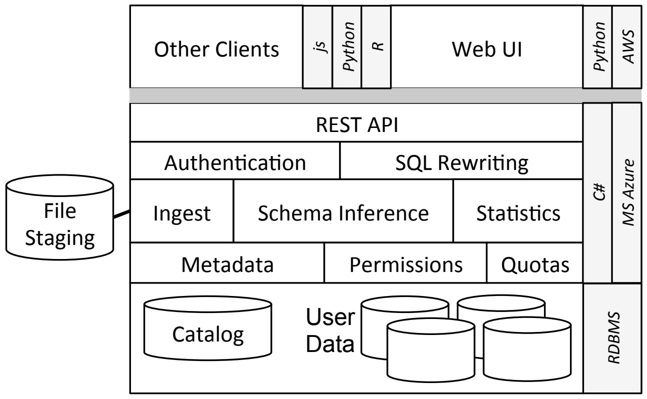


Figure 3: SQLShare architecture, primary user interface is the Web UI which communicates with the REST layer for dataset ingest, modification (via views) and querying. REST server interacts with the backend database which also keeps a catalog of user queries.

user based on their own requirements, and multiple javascript-based visualization interfaces have been developed.

3.5 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 [31] or Google Fusion Tables [12]. 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 [26] [21]), 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 grammar 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.

Figure 1 shows a screenshot of the SQL editor. 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 5.3.

4. 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 are releasing 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 4 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 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.

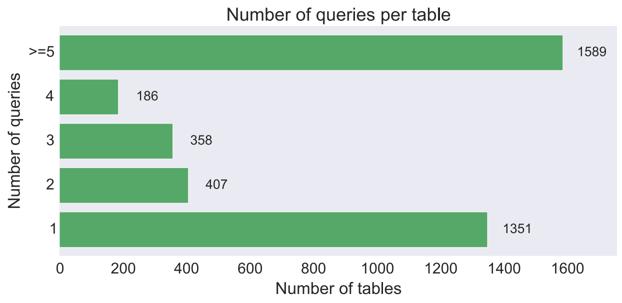


Figure 4: Distribution of queries per table. a) About a third of the tables accessed just once. b) Greater than a third of the tables are accessed many times, with the most common table being queried 766 times, suggesting two distinct use cases.

Extracting information from query logs.

To analyze the complexity and diversity of the SQLShare logs (Section 6), 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[2]. The operator tree along with interesting properties needed for further analysis is extracted from the XML document with XPath [6]. These properties are estimated 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 5a provides a conceptual visualization of Phase 1.

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

Users	591
Tables	3891
Columns	73070
Views	7958
Non-trivial Views	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 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 5b 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 and bundled all functionality as a python library whose source code is available online on demand [3]. As a sample implementation for other workloads, we have provided code to perform this analysis on SDSS and TPCH [8]. The python library also implements most of analysis that we make in the sections that follow.

Listing 1 Extracted structure and properties from a sample query.

```
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"
```

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 is shown in table Table 2a and Table 2b. As mentioned in §3.2 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]https://github.com/uwescience/query-workload-analysis

5. 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.

5.1 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. ii) *Post hoc Column Types*: 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.

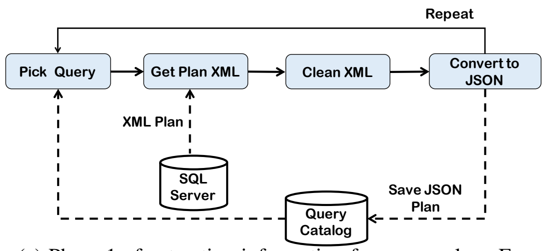
Vertical Recomposition: Datasets presented to SQLShare are often decomposed into multiple files that reflect the manner in which the data was collected. Rather than requiring that these files be concatenated offline or requiring that a single table be designed to store all such data, we encourage users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

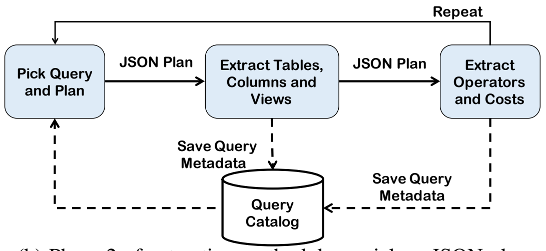
Overall, the data suggests that relaxed schemas played an important role in many use cases, and that tolerance for weakly structured data is an important part of any data system targeting science and data science environments.

5.2 Views Afford Controlled Data Sharing

The view-centric data model of SQLShare (Figure 2) 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 6 shows



(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 5: Workload Analysis Methodology

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[4].

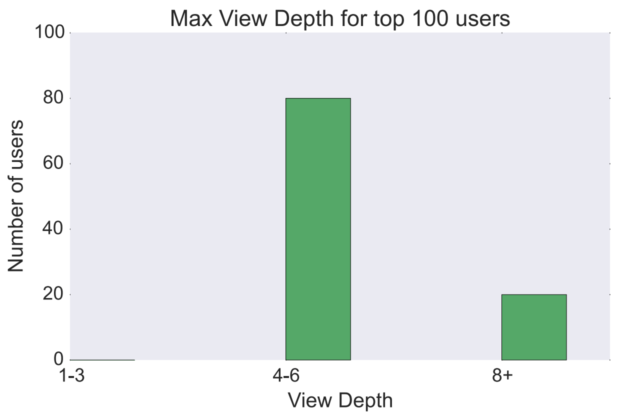


Figure 6: The maximum view depth for the 100 most active users of SQLShare. The data suggest that the ability to derive and save new datasets using views was an important features.

The view-centric data model also facilitates sharing: users can set daatset-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, 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.

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

5.3 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 [31], Google Fusion Tables [12]).

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 `*` 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."

6. 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) [18].

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 [28] 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.

6.1 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 7 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.

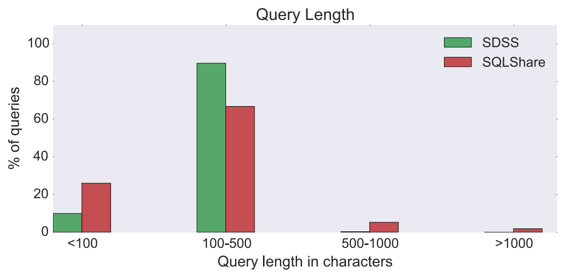


Figure 7: Hand-written SQLShare queries tend to vary more widely in length than SDSS queries. Short queries tend to be shorter, but long queries tend to be longer. SDSS queries show evidence of being "canned" rather than hand-written: only a few distinctive lengths are present and the majority of SDSS queries are about 200 characters. We explore these patterns in more detail in §6.2.

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 8 shows the number of distinct operators for the three workloads.

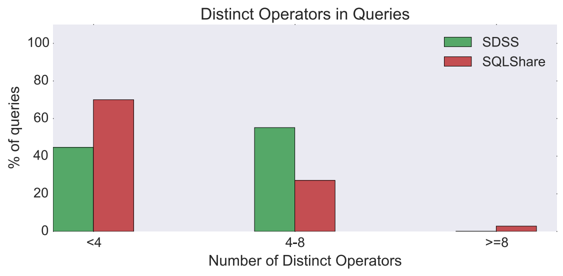


Figure 8: CDF of the number of distinct operators per query in both workloads. SQLShare has many queries with very few distinct operators, but about $5-8\%$ of the most complex queries have many more distinct operators than SDSS, suggesting that most complex queries in SQLShare appear to be more complex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

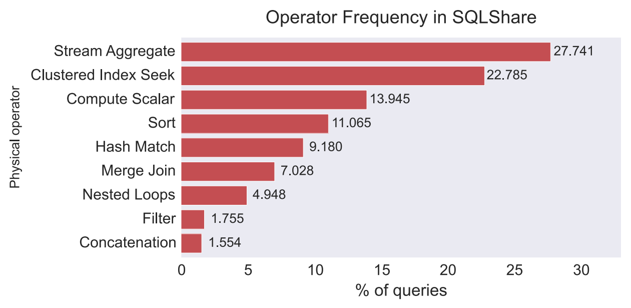


Figure 9: The most commonly used physical operators in SQLShare. We ignored Clustered Indexed Scan because SQLShare uses SQLAzure which requires them. Presence of a lot of aggregate and arithmetic operators in SQLShare suggests the presence of analytic workloads.

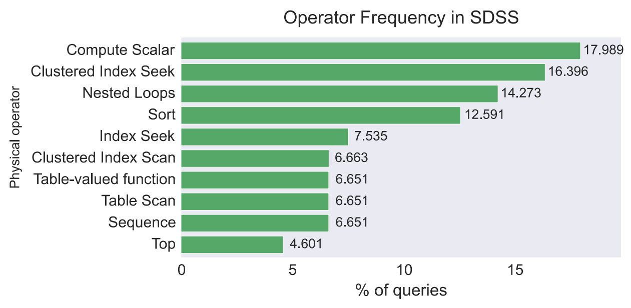


Figure 10: The most-used operator in SDSS is computation on scalars as a lot of queries use UDFs. Compared to SQLShare we see fewer arithmetic and aggregate operators.

Figure 9 and Figure 10 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 because it consists of a lot of user defined functions[5]. Overall, we find indications that users write more complex queries in SQLShare, suggesting that support for full SQL is useful for users.

6.2 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 a corpus of hand-written real queries is unavailable. For example, research on query recommendation platform like SnipSuggest [20] 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.* [23]: 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. While none of

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

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

these measures are 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 [6] and arithmetic expressions in SDSS are two scalar expressions followed by `BIT_AND`, `like` and `upper` (Table 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 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 and also include *data ingest, integration and cleaning* as suggested by the prevalence of string operations.

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
GetRange ThroughConvert	25746
GetRangeWith MismatchedTypes	25746
BIT_AND	21850
like	2376
upper	2312

(b) SDSS

Table 4: Most common intrinsic & arithmetic expression operators. String operations are very common on SQLShare, suggesting a lot of data integration and munging tasks.

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 [27]. Query optimization in the presence of cached results and materialized views is beyond the scope of this paper. 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.

6.3 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 11 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. 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 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.

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

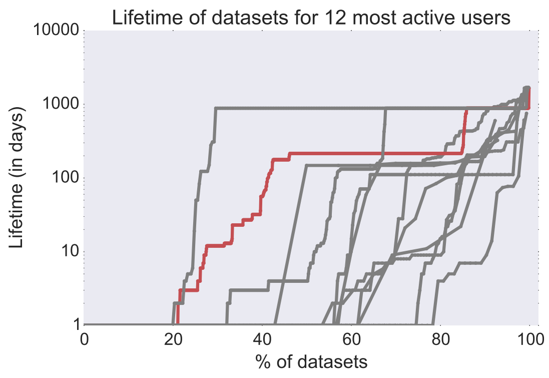


Figure 11: Dataset lifetimes for 12 most active users in SQLShare. Each curve is a user. The y-axis is the number of days between the first and last time the dataset was accessed. The x-axis is rank order. The great majority of datasets are accessed across a span of less 10 days, but some are accessed across periods of years. 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.

SQLShare users in general are varied in their patterns of data upload and data analysis. Figure 12 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.

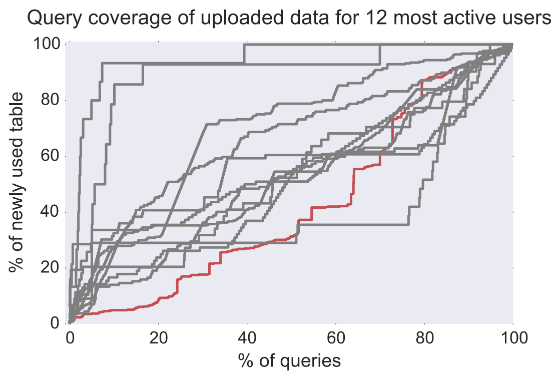


Figure 12: The rate of table coverage over time for the 12 most active users. Highlighted in red is the most active user for SQLShare. Each curve corresponds to a user and describes the percentage of tables accessed by the first N% of queries. A user who uploads one table at a time and queries it once 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.

6.4 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 13 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 [23]: 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 [23]. 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.

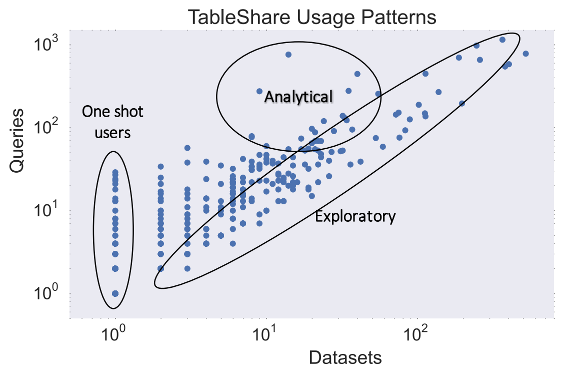


Figure 13: The ratio of datasets to queries suggests different usage patterns. The y-axis is the log of the number of queries, and the x-axis is the log of the number of datasets. Each point represents one user. Exploratory users only write a small number of queries over each dataset they upload. Some 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.

7. RELATED WORK

Structure extraction tools cast the data variety problem as one of parsing complex formats to produce (weakly) structured data for further processing. OpenRefine [3] and Wrangler [16] are examples of this approach. These tools offer no support for working with multiple datasets or managing complexity once the parsing step has been completed, which has been shown to be a dominant cost [17].

In traditional data integration approaches the central goal is to derive a mediated schema from two or more source schemas, allowing all source data to be queried uniformly [10]. Tools and algorithms in this area induce relationships between database elements (tables, columns, rows) and use these relationships to rewrite queries or restructure data. Despite a long history of research (and a detour through XML in the early part of this century), these techniques do not seem to be widely used by analysts today, in part because of the assumptions that the input schemas are carefully engineered, information carrying structures on which the algorithms can gain purchase.

A welcome departure from the engineered schemas assumed by data integration techniques was *dataspaces* [11].The paper was prescient in its ability to capture important aspects of the high variety problem, but focused heavily on enterprise settings and managed environments rather than the ad hoc, one-off analysis that characterizes data science activities we see in practice. It is unusual to assume that we would know all kind of operations and all sources of data before hand. This approach misses the point that in Big Data, data sources and the operations required are usually not known beforehand. However, in case of SQLShare, the departure from the above mentioned approaches is that it enables scientific data analysis with a bare minimum set of changes to the original RDBMS. Infact, at its core, SQLShare uses pure relational models. It just hides these from the users. Our work shows that how even with these basic changes, we can cater to the entire data lifecycle, *i.e.* ingest, cleaning, synthesis, management, analytics, visualization and sharing.

Khoussainova *et al.* noted in [19] that the mode of interaction with databases is changing. Data analytics warrants the support for exploratory queries, query recommendations and collaborative query management. We believe SQLShare is a right step in this direction. Bhardwaj *et al.* presented DataHub [5], a system which provides the collaborative data management and versioning, but unlike SQLShare it lacks data querying capabilities and support for full SQL.

SnipSuggest [20] is an example of a system which can enable non-experts to write complex SQL queries and even teach them. This work is complementary to one aspect of SQLShare *i.e.* reducing 'friction' for query authoring. It would be interesting to see how user query complexity changes overtime using SnipSuggest with SQLShare. Ogasawara *et al.* [24] presented support for automatic optimization data science workflows, motivating research in database support for scientific workloads. Recently, Fei *et al.* presented Schema-Free SQL [22] which tries to enables faster analysis of data for which the schema is unknown. This approach is different from the automatic schema inference in SQLShare and builds language support for inaccurate/unknown schema.

Ren *et al.* performed a workload analysis over 3 different Hadoop [1] research clusters [25]. 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 [28] 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. SQLShare takes this a step further by providing an SDSS like system for everyone. By enabling the user to upload datasets and enable sharing and dataset hierarchy, SQLShare unfolds a lot more avenues for easy scientific data management.

8. CONCLUSION

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. Finally, we gave a preliminary analysis of the workload 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.

The SQLShare system was designed to be minimal adaptation of existing database technology to attract users in ad hoc, high-touch analytics environments. We relaxed assumptions about schemas, inferring structure automatically from the data and tolerating various structural problems (mismatched row lengths, non-homogeneous types, missing column names). We adopt a view-oriented data model to encourage reuse, track provenance, simplify editing, and facilitate data sharing. We support full SQL, finding that impoverised SQL-like languages are not sufficient to express the queries in our workload. We found these features of SQLShare to be demonstrably useful by analyzing patterns in the workload. More generally, we conclude that SQLShare is useful for short lifetime, one-shot analysis tasks that are typically ill-suited for databases, and that these tasks produced queries that are more complex and more diverse.

To define query complexity, we considered character length of the query as an ASCII string and the number of distinct operators. We showed that by these measures SQLShare had attracted hand-written queries that are more complex than the most complex queries in the comparable Sloan Digital Sky Survey workload, another public query workload associated with a conventional database application in a comparable domain.

To define workload diversity, we considered the entropy of the workload: the number of unique queries divided by the number of total queries. To define query uniqueness, we considered string uniqueness as a naive baseline, but recognizing that two queries that have an identical structure but differ in literal values should not be considered truly distinct. We therefore considered a templatization procedure that unified the structure of queries in the log and created equivalence classes based on query patterns. By both measures of entropy, the SQLShare workload was more diverse than the SDSS workload. We also considered the use of expressions and scalar functions. While expressions were more varied in the SQLShare dataset, the SDSS workload involved a significant number of user-defined functions.

In future work, we plan to use this workload to provide a formal definition for query complexity and design languages and editors that can provably reduce complexity. We can use this definition to build more effective query recommendation engines which recommends queries of comparable complexity to queries that user has written before. We also plan to use the complexity and diversity properties of the query workload to design a formal benchmark emphasizing high variety rather than hig volume or high velocity. Our analysis will also inform the development of new systems such as Myria [13] that are designed to address analytics workloads in science and data science.

9. ACKNOWLEDGEMENTS

This project was supported by NSF IIS award # 1247469, the Intel Science and Technology Center for Big Data and the University of Washington eScience Institute, which made this work possible.

10. REFERENCES

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

[2] Big data techniques applied to media and computer graphics applications. https://metanautix.com/tr/01_big_data_techniques_for_media_graphics.pdf.

[3] OpenRefine (formerly google refine). http://openrefine.org/. Accessed: 2014-10-14.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[5] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798*, 2014.

[6] J. Clark, S. DeRose, et al. Xml path language (xpath). *W3C recommendation*, 16, 1999.

[7] S. Cohen-Boulakia and U. Leser. Search, adapt, and reuse: the future of scientific workflows. *ACM SIGMOD Record*, 40(2):6–16, 2011.

[8] T. P. P. Council. TPC-H benchmark specification. http://www.tpc.org/tpch/, 2008.

[9] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.

[10] A. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AI magazine*, 26(1):83, 2005.

[11] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *ACM Sigmod Record*, 34(4):27–33, 2005.

[12] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. 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.

[13] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, Sigmod '14, pages 881–884. ACM, 7 2014.

[14] B. Howe, G. Cole, E. Souroush, P. Koutris, A. Key, N. Khoussainova, and L. Battle. Database-as-a-service for long-tail science. In *Scientific and Statistical Database Management*, pages 480–489. Springer, 2011.

[15] B. Howe, F. Ribalet, D. Halperin, S. Chitnis, and E. V. Armbrust. Sqlshare: Scientific workflow via relational view sharing. *Computing in Science & Engineering, Special Issue on Science Data Management*, 15(2), 2013.

[16] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[17] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. In *IEEE Visual Analytics Science & Technology (VAST)*, 2012.

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

[19] N. Khoussainova, M. Balazinska, W. Gatterbauer, Y. Kwon, and D. Suciu. A case for a collaborative query management system. *arXiv preprint arXiv:0909.1778*, 2009.

[20] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for sql. *Proceedings of the VLDB Endowment*, 4(1):22–33, 2010.

[21] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[22] F. Li, T. Pan, and H. V. Jagadish. Schema-free sql. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1051–1062, New York, NY, USA, 2014. ACM.

[23] B. Mozafari, E. Z. Y. Goh, and D. Y. 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.

[24] E. Ogasawara, J. Dias, F. Porto, P. Valduriez, and M. Mattoso. An algebraic approach for data-centric scientific workflows. *Proc. of VLDB Endowment*, 4(12):1328–1339, 2011.

[25] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment*, 6(10):853–864, 2013.

[26] M. Rosson and J. Carroll. Active programming strategies in reuse. In O. Nierstrasz, editor, *ECOOPâĂŹ 93 âĂŤ Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 4–20. Springer Berlin Heidelberg, 1993.

[27] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan. Don't trash your intermediate results, cache'em. *arXiv preprint cs/0003005*, 2000.

[28] V. Singh, J. Gray, A. Thakar, A. S. Szalay, J. Raddick, B. Boroski, S. Lebedeva, and B. Yanny. Skyserver traffic report-the first five years. *arXiv preprint cs/0701173*, 2007.

[29] M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and scidb. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[30] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer Publishing Company, Incorporated, 2014.

[31] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.