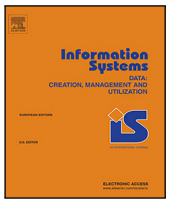
Information Systems 138 (2026) 102687

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Example-driven semantic-similarity-aware query intent discovery: Empowering users to cross the SQL barrier through query by example☆

Anna Fariha [a,*], Lucy Cousins [b], Narges Mahyar [c], Alexandra Meliou [c,d]

[a] University of Utah, Salt Lake City, UT, USA
[b] Liberty Mutual Insurance, Boston, MA, USA
[c] University of Massachusetts, Amherst, MA, USA
[d] Archimedes, Athena RC, Greece

ARTICLE INFO

Keywords:
Query by example
Abductive reasoning
User studies

ABSTRACT

Traditional relational data interfaces require precise structured queries over potentially complex schemas. These rigid data retrieval mechanisms pose hurdles for nonexpert users, who typically lack programming language expertise and are unfamiliar with the details of the schema. Existing tools assist in formulating queries through keyword search, query recommendation, and query auto-completion, but still require some technical expertise. An alternative method for accessing data is *query by example* (QBE), where users express their data exploration intent simply by providing examples of their intended data and the system infers the intended query. However, existing QBE approaches focus on the structural similarity of the examples and ignore the richer context present in the data. As a result, they typically produce queries that are too general, and fail to capture the user's intent effectively. In this article, we present SQuID, a system that performs *semantic-similarity-aware* query intent discovery from user-provided example tuples.

Our work makes the following contributions: (1) We design SQuID: an end-to-end system that automatically formulates select-project-join queries with optional group-by aggregation and intersection operators – a much larger class than what prior QBE techniques support – from user-provided examples, in an open-world setting. (2) We express the problem of query intent discovery using a *probabilistic abduction model* that infers a query as the most likely explanation of the provided examples. (3) We introduce the notion of an *abduction-ready* database, which precomputes semantic properties and related statistics, allowing SQuID to achieve real-time performance. (4) We present an extensive empirical evaluation on three real-world datasets, including user intent case studies, demonstrating that SQuID is efficient and effective, and outperforms machine learning methods, as well as the state of the art in the related query reverse engineering problem. (5) We contrast SQuID with traditional SQL querying through a comparative user study, which demonstrates that users with varying expertise are significantly more effective and efficient with SQuID than SQL. We find that SQuID eliminates the barriers in studying the database schema, formalizing task semantics, and writing syntactically correct SQL queries, and, thus, substantially alleviates the need for technical expertise in data exploration.

1. Introduction

The proliferation of computational resources and data sharing platforms has reached an ever-growing base of users without technical computing expertise, who wish to peruse, analyze, and understand data. From astronomers and scientists who need to analyze data to validate their hypotheses, all the way to computational journalists who need to peruse datasets to validate claims and support their reporting, the broad availability of data has the potential to fundamentally impact the way domain experts conduct their work. Unfortunately, while data is broadly available, data access is seldom unfettered. Existing systems typically cater to users with sound technical computing and programming skills, posing significant hurdles to technical novices, who do not have strong technical background. *Democratization* of computational systems demands equal access to people of different skills and backgrounds [1,2].

☆ This article is part of a Special issue entitled: 'HILDA' published in Information Systems.

* Corresponding author.
E-mail addresses: afariha@cs.utah.edu (A. Fariha), lucyrcousins@gmail.com (L. Cousins), nmahyar@cs.umass.edu (N. Mahyar), ameli@cs.umass.edu (A. Meliou).
URLs: https://afariha.github.io/ (A. Fariha), https://groups.cs.umass.edu/nmahyar/ (N. Mahyar), https://people.cs.umass.edu/~ameli/ (A. Meliou).

https://doi.org/10.1016/j.is.2026.102687
Received 12 June 2025; Received in revised form 8 November 2025; Accepted 7 January 2026
Available online 12 January 2026

academics

id	name
100	Thomas Cormen
101	**Dan Suciu**
102	Jiawei Han
103	**Sam Madden**
104	James Kurose
105	Joseph Hellerstein

research

aid	interest
100	algorithms
101	data management
102	data mining
103	data management
103	distributed systems
104	computer networks
105	data management
105	distributed systems

Fig. 1. Excerpt of two relations of the `CS Academics` database. `Dan Suciu` and `Sam Madden` (in bold), both have research interests in `data management`.

Like many computational systems, traditional database technology was not designed with the group of nonexpert users in mind, and, hence, poses hurdles to them. Traditional query interfaces allow data retrieval through well-structured queries, and to write such queries, one needs expertise in the query language (typically SQL) and knowledge of the potentially complex database schema. Unfortunately, nonexpert users typically lack both. Example-based interactions have been explored as a method to bridge the usability gap of computational systems that typically require precise programs from users. Under the *programming by example* (PBE) paradigm (also known as *programming by demonstration*), instead of writing a precise program to specify their intent, users only need to provide a few examples of the mechanism or result they desire [3–6].

Example-driven interactions have also been explored in the context of retrieving and exploring relational data, which led to the development of *query by example* (QBE) systems [7–9]. QBE offers an alternative data retrieval mechanism, where users specify their intent by providing example tuples for their query output [10]. Unfortunately, traditional QBE systems [7–9] for relational databases make a strong and oversimplified assumption in modeling user intent: they implicitly treat the structural similarity and data content of the example tuples as the only factors specifying query intent. As a result, they consider all queries that contain the provided example tuples in their result set as equally likely to represent the desired intent. This ignores the richer context in the data that can help identify the intended query more accurately. While more nuanced QBE systems exist, they typically place additional requirements or significant restrictions over the supported queries (Fig. 3 & Section 10.2).

Example 1.1. In Fig. 1, the relations `academics` and `research` store information about CS researchers and their research interests. Given the user-provided set of examples `{Dan Suciu, Sam Madden}`, a human can posit that the user is likely looking for researchers in the area of `data management`. However, a QBE system that looks for queries only based on the structural similarity of the examples produces `Q1` to capture the query intent, which is too general:

```
Q1: SELECT name FROM academics
```

In fact, the QBE system will generate the same generic query `Q1` for any set of names from the relation `academics`. Even though the intended semantic context is present in the data (by associating academics with research interest information using the relation `research`), existing QBE systems fail to capture it. A more specific query that better represents the semantic similarity among the example tuples is `Q2`:

```
Q2: SELECT name FROM academics, research
    WHERE research.aid = academics.id
    AND research.interest = 'data management'
```

Example 1.1 shows how reasoning about the semantic similarity of the example tuples can guide the discovery of the correct query structure (join of the `academics` and `research` tables), as well as the discovery of the likely intent (research interest in `data management`).

We can often capture semantic similarity through direct attributes of the example tuples. These are attributes associated with a tuple within the same relation, or through simple key-foreign key joins (such as research interest in Example 1.1). Direct attributes capture intent that is *explicit*, precisely specified by the particular attribute values. However, sometimes query intent is more vague, and is not expressible by explicit semantic similarity alone. In such cases, the semantic similarity of the example tuples is *implicit*, which can be captured through deeper associations with other entities in the data (e.g., genre and number of movies an actor appears in).

Example 1.2. The IMDb dataset contains a wealth of information related to the film and entertainment industry. We query the IMDb dataset (Fig. 2) with a traditional QBE system (e.g., [7]), using two different sets of examples:

```
ET1 = {Arnold Schwarzenegger,          ET2 = {Eddie Murphy,
       Sylvester Stallone,                    Jim Carrey,
       Dwayne Johnson}                        Robin Williams}
```

`ET1` contains the names of three actors from a public list of "physically strong" actors[1]; `ET2` contains the names of three actors from a public list of "funny" actors.[2] `ET1` and `ET2` represent different query intents ("strong" actors and "funny" actors, respectively), but a standard QBE system produces the same generic query for both:

```
Q3: SELECT person.name FROM person
```

Explicit semantic similarity cannot capture these different intents, as there is no attribute that explicitly characterizes an actor as "strong" or "funny". Nevertheless, the database encodes these associations implicitly, in the number and genre of movies an actor appears in ("strong" actors frequently appear in action movies, and "funny" actors in comedies).

Standard QBE systems typically produce queries that are too general, and, thus, fail to capture nuanced query intents, such as the ones in Examples 1.1 and 1.2. Some prior approaches attempt to refine the queries based on additional external information, such as external ontologies [11], provenance information of the example tuples [9], and user feedback on multiple (typically a large number of) system-generated examples [12–14]. Other work relies on a *closed-world* assumption – where a tuple not specified as an example output is assumed to be excluded from the query result – to produce more expressive queries [13,15,16], and, thus, requires complete examples of input databases and output results. Providing such external information is typically complex and tedious for nonexperts.

In contrast with prior approaches, in this article, we propose a method and present an end-to-end system for discovering query intent effectively and efficiently, in an open-world setting, without the

[1] Physically strong actors: https://www.imdb.com/list/ls050159844

[2] Funny actors: https://www.imdb.com/list/ls000025701

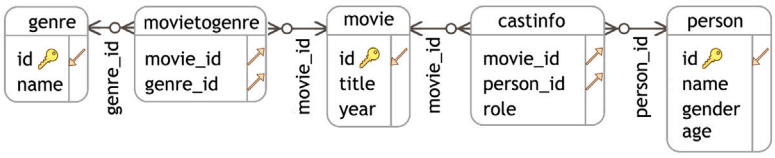


Fig. 2. Partial schema of the IMDb database. The schema contains two entity relations: `movie` and `person`; and a semantic property relation: `genre`. The relations `castinfo` and `movietogenre` associate entities and semantic properties.

Legend			query class								
QBE: Query by Example QRE: Query Reverse Engineering DX: Data Exploration KG: Knowledge Graph ◐: with significant restrictions			join	projection	selection	aggregation	semi-join	implicit property	scalable	open-world	additional requirements
QBE	relational	**SQuID**	●	●	●	●	●	●	●	●	
		Bonifati et al. [12]	●	●	◐		●		●	●	user feedback
		QPlain [9]	●	●	●		●		●	●	provenance input
		Shen et al. [7]	●	●					●	●	
		FastTopK [8]	●	●					●	●	
	KG	Arenas et al. [54]	●	●	◐		●		●	●	
		SPARQLByE [53]	●	●	◐		●		●	●	negative examples
		GQBE [19]	●	●	◐		●		●	●	
		QBEES [18]	●	●	◐		●		●	●	
QRE	relational	PALEO-J [61]	●	●	●	●			●		top-k queries only
		SQLSynthesizer [16]	●	●	●	●	●				schema knowledge
		Scythe [15]	●	●	●	●	●				schema knowledge
		Zhang et al. [30]	●						●	●	
		REGAL [60]		●	●	●			●		
		REGAL+ [62]	●	●	●	●					
		FastQRE [29]	●	●					●	●	
		QFE [13]	●	●	●						user feedback
		TALOS [20]	●	●	●	●	●		●		
DX	rel.	AIDE [14]			●				●	●	user feedback
		REQUEST [57]			●				●	●	user feedback

Fig. 3. SQuID captures complex intents and more expressive queries than prior work in the open-world setting. Details are in Section 10.2.

need for any additional external information beyond the initial set of example tuples. While Fig. 3 provides a summary exposition of prior work, and contrasts with our contributions, we detail this classification and metrics and discuss the related work in Section 10. SQuID, our semantic-similarity-aware query intent discovery framework, relies on two key insights: (1) It exploits the information and associations already present in the data to derive the explicit and implicit similarities among the provided examples. (2) It identifies the significant semantic similarities among them using *abductive reasoning*, a logical inference mechanism that aims to derive a query as the simplest and most likely explanation of the observation (example tuples). We proceed to explain how SQuID uses these insights to handle the challenging scenario of Example 1.2 next.

Example 1.3. We query the IMDb dataset with SQuID, using the example tuples of `ET2` (Example 1.2). SQuID discovers the following semantic similarities among the examples: (1) all are `Male`, (2) all are `American`, and (3) all appeared in more than 40 `Comedy` movies. Out of these properties, `Male` and `American` are very common in the IMDb database. In contrast, a very small fraction of persons in the dataset are associated with such a high number of `Comedy` movies; this means that it is unlikely for this similarity to be coincidental, as opposed to the other two. Based on abductive reasoning, SQuID selects the third semantic similarity as the best explanation of the observed example tuples, and produces the query:

```
Q4: SELECT person.name
    FROM person, castinfo, movietogenre, genre
    WHERE person.id = castinfo.person_id AND
        castinfo.movie_id = movietogenre.movie_id AND
        movietogenre.genre_id = genre.id AND genre.name =
      'Comedy'
    GROUP BY person.id
    HAVING count(*) >= 40
```

In this article, we make the following contributions:

- We design SQuID: an end-to-end system that automatically formulates select-project-join queries with optional group-by aggregation and intersection operators (SPJ_{AI}) based on few user-provided example tuples (Section 2). SQuID does not require the users to have any knowledge of the database schema or the query language. Unlike existing approaches, SQuID does not require any additional information from the user, beyond the example tuples.
- SQuID infers *semantic similarities* of the examples and models query intent using a collection of *basic* and *derived* semantic property *filters* (Section 3). While some prior work explored the use of semantic similarity in knowledge graphs [17–19], they do not directly apply to the relational domain, as they do not model implicit semantic similarities derived from aggregating properties of affiliated entities (e.g., number of comedy movies an actor appears in).
- We express the problem of query intent discovery using a *probabilistic abduction model* (Section 4). This model allows SQuID to identify the semantic property filters that represent the most likely intent, given the examples.

- SQuID achieves real-time performance through an offline strategy that precomputes semantic properties and related statistics to construct an *abduction-ready* database (Section 5). During the online phase, SQuID consults the abduction-ready database to derive relevant semantic property filters, based on the provided examples, and applies abduction to select the optimal set of filters towards query intent discovery (Section 6).
- Our empirical evaluation includes three real-world datasets, 41 queries covering a broad range of complex intents and structures, and three case studies (Section 7). We further compare SQuID with TALOS [20], a state-of-the-art query reverse engineering system that supports very expressive queries, but in a closed-world setting. We show that SQuID is more accurate at capturing intents and infers better queries, often reducing the number of predicates by orders of magnitude. We also empirically show that SQuID outperforms a semi-supervised positive and unlabeled learning system [21].
- We present results of two comparative user studies – a controlled experiment study and an interview study – contrasting SQuID with the traditional SQL querying (Section 8). Our analysis of the controlled experiment study shows that participants were significantly more *effective* (achieved more accurate results) and *efficient* (required less time and fewer attempts) over a diverse set of data-exploration tasks using SQuID than SQL. Qualitative feedback of the interviewees confirms that SQuID eliminates SQL challenges and assists the users in effective data exploration. While our results validate some findings of prior studies over other PBE approaches [22], we contribute new empirical insights gained from our studies that indicate that even a limited level of domain expertise (knowledge of a small subset of the desired data) can substantially help overcome the lack of technical expertise (knowledge of SQL and schema) in data exploration.
- Finally, in light of the findings from the comparative user studies, we identify three key challenges that SQL poses to the users: (1) familiarizing oneself with the database schema, (2) formally expressing the semantics of the tasks, and (3) writing syntactically correct queries; and discuss how SQuID can effectively eliminate these challenges. We further discuss how SQuID and traditional SQL interface complement each other, under what circumstances the users prefer one over the other, and how the QBE tools should be expanded to achieve more user acceptance (Section 9).

2. SQuID overview

In this section, we first discuss the challenges in example-driven query intent discovery and highlight the shortcomings of existing approaches. We then formalize the problem of query intent discovery using a probabilistic model and describe how SQuID infers the most likely query intent using abductive reasoning. Finally, we present the system architecture for SQuID, and provide an overview of our approach.

2.1. The query intent discovery problem

SQuID tackles three key limitations of existing QBE systems:

Large search space. Identifying the intended query, given a set of example tuples, can involve a huge search space of potential candidate queries. Aside from enumerating the candidate queries, validating them is expensive, as it requires executing the queries over potentially very large data. Existing approaches limit their search space in three ways: (1) They often focus on project-join (PJ) queries only. Unfortunately, ignoring selections severely limits the applicability and practical impact of these solutions. (2) They assume that the user provides a large number of examples or interactions, which is often unreasonable in practice. (3) They make a closed-world assumption, thus needing complete sets of input data and output results. In contrast, SQuID focuses on a much larger and more expressive class of queries, *select-project-join queries with optional group-by aggregation and intersection operators* (SPJ_{AI}),[3] and is effective in the open-world setting with very few examples.

Distinguishing candidate queries. In most cases, a set of example tuples does not uniquely identify the target query, i.e., there are multiple valid queries that contain the example tuples in their results. Most existing QBE systems do not distinguish among the valid queries [7] or only rank them according to the degree of input containment, when the example tuples are not fully contained by the query output [8]. In contrast, SQuID exploits the semantic context of the example tuples and ranks the valid queries based on a probabilistic abduction model of query intent.

Complex intent. A user's information need is often more complex than what is explicitly encoded in the database schema (e.g., Example 1.2). Existing QBE solutions focus on the query structure, and, thus, are ill-equipped to capture nuanced intents. While SQuID still produces a structured query in the end, its objectives focus on capturing the semantic similarity of the examples, both explicit and implicit. SQuID thus draws a contrast between the traditional query-by-example problem, where the query is assumed to be the hidden mechanism behind the provided examples, and the *query intent discovery problem*, which is our focus.

We proceed to formalize the problem of query intent discovery. We use $\mathcal{D}$ to denote a database, and $Q(\mathcal{D})$ to denote the set of tuples in the result of query Q operating on $\mathcal{D}$.

Definition 2.1 (*Query Intent Discovery*)**.** For a database $\mathcal{D}$ and a user-provided example tuple set E, the query intent discovery problem is to find an SPJ_{AI} query Q such that:

- $E \subseteq Q(\mathcal{D})$
- $Q = \text{argmax}_q \; Pr(q \mid E)$

More informally, we aim to discover an SPJ_{AI} query Q that contains E within its result set and maximizes the query posterior, i.e., the conditional probability $Pr(Q \mid E)$.

2.2. Abductive reasoning

SQuID solves the query intent discovery problem (Definition 2.1) using *abduction*. Abduction or abductive reasoning [23–26] refers to the method of inference that finds the best explanation (query intent) of an often incomplete observation (example tuples). Unlike deduction, in abduction, the premises do not guarantee the conclusion. A deductive approach would produce all possible queries that contain the example tuples in their results, guaranteeing that the intended query is one of them. However, the set of valid queries can be extremely large, growing exponentially with the number of properties and the size of the data domain. Hence, we model query intent discovery as an abduction problem and apply abductive inference to discover the most likely query intent. For example, given two possible candidate queries, Q and Q', we infer Q as the intended query if $Pr(Q \mid E) > Pr(Q' \mid E)$.

Example 2.1. In the scenario of Example 1.1, SQuID identifies that the two example tuples share the semantic context `interest = 'data management'`. While `Q1` and `Q2` both contain the examples tuples in their result set, the probability that two tuples drawn randomly from the output of `Q1` would display the identified semantic context is low ($(\frac{3}{7})^2 \approx 0.18$ in the data excerpt). In contrast, the probability that two tuples drawn randomly from the output of `Q2` would display the same semantic context is high (1.0). Assuming equal priors for `Q1` and `Q2`, from Bayes' rule: $Pr(Q2 \mid E) > Pr(Q1 \mid E)$.

[3] The SPJ_{AI} queries derived by SQuID limit joins to key-foreign key joins, and conjunctive selection predicates of the form `attribute OP value`, where `OP` $\in \{=, \geq, \leq\}$ and `value` is a constant.

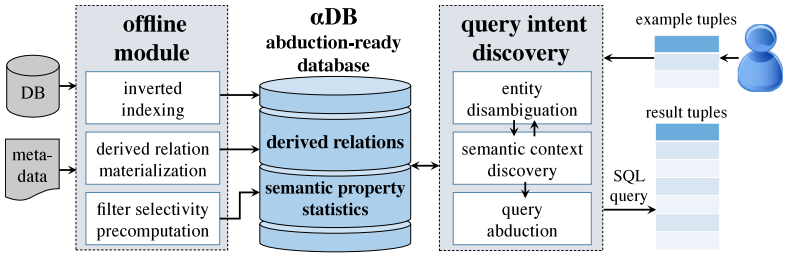


Fig. 4. SQuID's operation includes an offline module, which constructs an *abduction-ready* database (αDB) and precomputes statistics of semantic properties. During normal operation, SQuID's query intent discovery module interacts with the αDB to identify the semantic context of the user-provided example tuples and abduces the most likely query intent.

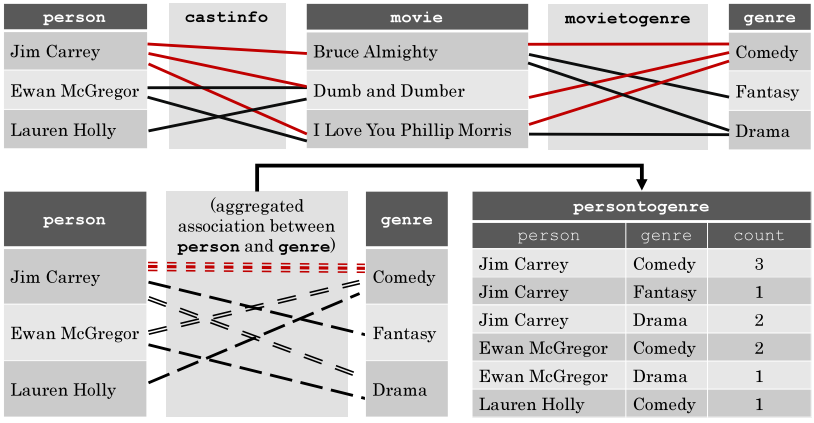


Fig. 5. A `genre` value (e.g., `genre=Comedy`) is a basic semantic property of a `movie` (through the `movietogenre` relation). A `person` is associated with `movie` entities (through the `castinfo` relation); aggregates of basic semantic properties of movies are *derived semantic properties* of `person`, e.g., the number of comedy movies a person appeared in. The αDB stores the derived property in the new relation `persontogenre`. (For ease of exposition, we depict attributes `genre` and `person` instead of `genre.id` and `person.id`.)

2.3. Solution sketch

At the core of SQuID is an *abduction-ready database*, αDB (Fig. 4). The αDB serves two purposes: (1) it increases SQuID's efficiency by storing precomputed associations and statistics, and (2) it simplifies the query model by reducing the extended family of SPJ_{AI} queries on the original database to equivalent SPJ queries on the αDB.

Example 2.2. The IMDb database has, among others, relations `person` and `genre` (Fig. 2). SQuID's αDB stores a derived semantic property that associates the two entity types in a new relation, `persontogenre(person.id, genre.id, count)`, which stores the number of movies of each genre each person appeared in. SQuID derives this relation through joins with `castinfo` and `movietogenre`, followed by aggregation (Fig. 5). Then, the SPJ_{AI} query `Q4` (Example 1.3) is equivalent to the simpler SPJ query `Q5` on the αDB:

```
Q5: SELECT person.name
    FROM person, persontogenre, genre
    WHERE person.id = persontogenre.person_id AND
        persontogenre.genre_id = genre.id AND
        genre.name = 'Comedy' AND persontogenre.count >= 40
```

By incorporating aggregations in precomputed, derived relations, SQuID can reduce SPJ_{AI} queries on the original data to SPJ queries on the αDB. SQuID starts by inferring a PJ query, Q^*, on the αDB as a query template; it then augments Q^* with selection predicates, driven by the semantic similarity of the examples.

Organization. Section 3 formalizes SQuID's model of query intent as a combination of the base query Q^* and a set of semantic property filters. Section 4 analyzes the probabilistic abduction model that SQuID uses to solve the query intent discovery problem (Definition 2.1). Then we describe SQuID's system components. Section 5 describes the offline module, which is responsible for making the database abduction-ready, by precomputing semantic properties and statistics in derived relations. Section 6 describes the query intent discovery module, which abduces the most likely intent as an SPJ query on the αDB.

3. Modeling query intent

SQuID's core task is to infer the proper SPJ query on the αDB. We model an SPJ query as a pair of a base query and a set of semantic property filters: $Q^{\varphi}=(Q^*, \varphi)$. The *base query* Q^* is a project-join query that captures the structural aspect of the example tuples. SQuID can handle examples with multiple attributes, but, for ease of exposition, we focus on example tuples that contain a single attribute of a single entity (`name` of `person`). In contrast to existing approaches that derive PJ queries from example tuples, the base query in SQuID does not need to be minimal with respect to the number of joins: while a base query on a single relation with projection on the appropriate attribute (e.g., `Q1` in Example 1.1) would capture the structure of the examples, the semantic context may rely on other relations (e.g., `research`, as in `Q2` of Example 1.1). Thus, SQuID considers any number of joins among αDB relations for the base query, but limits these to key-foreign-key joins. We discuss a simple method for deriving the base query in Section 6.2. SQuID's core challenge is to infer φ, which denotes a set of *semantic property filters* that are added as conjunctive selection predicates to Q^*. The base query and semantic property filters for `Q2` of Example 1.1 are:

Q^* = `SELECT name FROM academics, research`
`WHERE research.aid = academics.id`
φ = { `research.interest = 'data management'` }

Notation	Description
$p = \langle A, V, \theta \rangle$	Semantic property defined by attribute A, value V, and association strength θ
ϕ_p or ϕ	Semantic property filter for p
$\Phi = \{\phi_1, \phi_2, \dots\}$	Set of minimal valid filters
$Q^\varphi = (Q^*, \varphi)$	SPJ query with semantic property filters $\varphi \subseteq \Phi$ applied on base query Q^*
$x = (p, \lvert E \rvert)$	Semantic context of E for p
$\mathcal{X} = \{x_1, x_2, \dots\}$	Set of semantic contexts

Fig. 6. Summary of notations.

Sample database

id	name	gender	age
1	Tom Cruise	Male	50
2	Clint Eastwood	Male	90
3	Tom Hanks	Male	60
4	Julia Roberts	Female	50
5	Emma Stone	Female	29
6	Julianne Moore	Female	60

Example tuples

Column 1
Tom Cruise
Clint Eastwood

Fig. 7. Sample database (left) with example tuples (right).

3.1. Semantic properties and filters

Semantic properties encode characteristics of an entity. We distinguish semantic properties into two types. (1) A *basic semantic property* is affiliated with an entity directly. In the IMDb schema of Fig. 2, `"gender = Male"` is a basic semantic property of a `person`. (2) A *derived semantic property* of an entity is an aggregate over a basic semantic property of an associated entity. In Example 2.2, the number of movies of a particular genre that a person appeared in is a derived semantic property for `person`. We represent a semantic property p of an entity from a relation R as a triple $p = \langle A, V, \theta \rangle$. In this notation, V denotes a value or a value range for attribute A associated with entities in R.[4] The *association strength* parameter θ quantifies how strongly an entity is associated with the property. It corresponds to a threshold on derived semantic properties (e.g., the number of comedies an actor appeared in); it is not defined for basic properties ($\theta = \perp$). In this work, we consider association strength to be an absolute number. However, an alternative is to consider it as a relative number (e.g., ratio of comedy movies compared to total movies).

A *semantic property filter* ϕ_p is a structured language representation of the semantic property p. In Fig. 7, the filters $\phi_{\langle \texttt{gender},\texttt{Male},\perp \rangle}$ and $\phi_{\langle \texttt{age},[50,90],\perp \rangle}$ represent two basic semantic properties on `gender` and `age`, respectively. Expressed in relational algebra, filters on basic semantic properties map to standard selection predicates, e.g., $\sigma_{\texttt{gender=Male}}(\texttt{person})$ and $\sigma_{50 \le \texttt{age} \le 90}(\texttt{person})$. For derived properties, filters specify conditions on the association across different entities. In Example 2.2, for `person` entities, the filter $\phi_{\langle \texttt{genre},\texttt{Comedy},30 \rangle}$ denotes the property of a person being associated with at least 30 movies with the basic property `"genre = Comedy"`. In relational algebra, filters on derived properties map to selection predicates over derived relations in the αDB, e.g., $\sigma_{\texttt{genre=Comedy} \wedge \texttt{count} \ge 30}(\texttt{persontogenre})$.

3.2. Filters and example tuples

To construct Q^φ, SQuID needs to infer the proper set of semantic property filters given a set of example tuples. Since all example tuples should be in the result of Q^φ, φ cannot contain filters that the example tuples do not satisfy. Thus, we only consider *valid* filters that map to selection predicates that *all* example tuples satisfy.

Definition 3.1 (*Filter validity*)**.** Given a database D, an example tuple set E, and a base query Q^*, a filter ϕ is valid if $Q^{\{\phi\}}(D) \supseteq E$, where $Q^{\{\phi\}} = (Q^*, \{\phi\})$.

Example 3.1. Fig. 7 shows a set of example tuples over the relation `person`. Given the base query Q^* = `SELECT name FROM person`, the filters $\phi_{\langle \texttt{gender},\texttt{Male},\perp \rangle}$ and $\phi_{\langle \texttt{age},[50,90],\perp \rangle}$ on relation `person` are valid, because all of the example entities of Fig. 7 are `Male` and fall in the age range [50, 90].

Lemma 3.1 (*Validity of Conjunctive Filters*)**.** *The conjunction $(\phi_1 \wedge \phi_2 \wedge \dots)$ of a set of filters $\Phi = \{\phi_1, \phi_2, \dots\}$ is valid, i.e., $Q^\Phi(D) \supseteq E$, if and only if $\forall \phi_i \in \Phi$ ϕ_i is valid.*

Example 3.2. If $\phi_{\langle \texttt{age},[50,90],\perp \rangle}$ is valid, then $\phi_{\langle \texttt{age},[40,120],\perp \rangle}$ is also valid.

The above example shows that relaxing a filter (loosening its conditions) preserves validity. Among all valid filters, SQuID focuses on *minimal* valid filters, which have the tightest bounds. Bounds can be derived in other ways, e.g., informed by the result cardinality. However, we found the choice of the tightest bounds to work well in practice.

Definition 3.2 (*Filter Minimality*)**.** A basic semantic property filter $\phi_{\langle A,V,\perp \rangle}$ is *minimal* if it is valid, and $\forall V' \subset V$ $\phi_{\langle A,V',\perp \rangle}$ is invalid. A derived semantic property filter $\phi_{\langle A,V,\theta \rangle}$ is *minimal* if it is valid, and $\forall \epsilon > 0$ $\phi_{\langle A,V,\theta+\epsilon \rangle}$ is invalid.

For Fig. 7, $\phi_{\langle \texttt{age},[50,90],\perp \rangle}$ is a minimal filter and $\phi_{\langle \texttt{age},[40,90],\perp \rangle}$ is not.

4. Probabilistic abduction model

We now revisit the problem of query intent discovery (Definition 2.1), and recast it based on our model of query intent (Section 3). Specifically, Definition 2.1 aims to discover an SPJ_{AI} query Q, which is then reduced to an equivalent SPJ query Q^φ on the αDB (as in Example 2.2). SQuID's task is to find the query Q^φ that maximizes the posterior probability $Pr(Q^\varphi \mid E)$, for a given set E of example tuples. In this section, we analyze the probabilistic model to compute this posterior, and break it down to three components.

[4] SQuID can support disjunction for categorical attributes (e.g., `"gender = Male" OR "gender = Female"`), so V could be a set of values. However, for ease of exposition we keep our examples limited to properties without disjunction.

4.1. Notations and preliminaries

Semantic context x. Observing a semantic property in a set of 10 examples is more significant than observing the same property in a set of 2 examples. We denote this distinction with the *semantic context* $x = (p, |E|)$, which encodes the size of the set $|E|$, where the semantic property p is observed. We denote with $\mathcal{X} = \{x_1, x_2, \dots\}$ the set of semantic contexts exhibited by the set of example tuples E.

Candidate SPJ query Q^{φ}. Let $\Phi = \{\phi_1, \phi_2, \dots\}$ be the set of minimal valid filters, from hereon simply referred to as filters, where ϕ_i encodes the semantic context x_i. We omit $\langle A, V, \theta\rangle$ in the filter notation when the context in clear. Our goal is to identify the subset of filters in Φ that best captures the query intent. A set of filters $\varphi \subseteq \Phi$ defines a candidate query $Q^{\varphi} = (Q^*, \varphi)$, and $Q^{\varphi}(D) \supseteq E$ (from Lemma 3.1).

Filter event $\widetilde{\phi}$. A filter $\phi \in \Phi$ may or may not appear in a candidate query Q^{φ}. With slight abuse of notation, we denote the filter's presence ($\phi \in \varphi$) with ϕ and its absence ($\phi \notin \varphi$) with $\bar{\phi}$. We use $\widetilde{\phi}$ to represent the occurrence event of ϕ in Q^{φ}.

Thus: $\widetilde{\phi} = \begin{cases} \phi & \text{if } \phi \in \varphi \\ \bar{\phi} & \text{if } \phi \notin \varphi \end{cases}$

4.2. Modeling query posterior

We first analyze the probabilistic model for a *fixed base query* Q^* and then generalize the model in Section 4.3. We use $Pr_*(a)$ as a shorthand for $Pr(a \mid Q^*)$. We model the query posterior $Pr_*(Q^{\varphi} \mid E)$, using Bayes' rule:

$$Pr_*(Q^{\varphi} \mid E) = \frac{Pr_*(E \mid Q^{\varphi}) \cdot Pr_*(Q^{\varphi})}{Pr_*(E)}$$

By definition, $Pr_*(\mathcal{X} \mid E) = 1$; therefore:

$$Pr_*(Q^{\varphi} \mid E) = \frac{Pr_*(E, \mathcal{X} \mid Q^{\varphi}) \cdot Pr_*(Q^{\varphi})}{Pr_*(E)} = \frac{Pr_*(E \mid \mathcal{X}, Q^{\varphi}) Pr_*(\mathcal{X} \mid Q^{\varphi}) \cdot Pr_*(Q^{\varphi})}{Pr_*(E)}$$

Using the fact that $Pr_*(\mathcal{X} \mid E) = 1$ and applying Bayes' rule on the prior $Pr_*(E)$, we get:

$$Pr_*(Q^{\varphi} \mid E) = \frac{Pr_*(E \mid \mathcal{X}, Q^{\varphi}) Pr_*(\mathcal{X} \mid Q^{\varphi}) \cdot Pr_*(Q^{\varphi})}{Pr_*(E \mid \mathcal{X}) \cdot Pr_*(\mathcal{X})}$$

Finally, E is conditionally independent of Q^{φ} given the semantic context $\mathcal{X}$, i.e., $Pr_*(E \mid \mathcal{X}, Q^{\varphi}) = Pr_*(E \mid \mathcal{X})$. Thus:

$$Pr_*(Q^{\varphi} \mid E) = \frac{Pr_*(\mathcal{X} \mid Q^{\varphi}) \cdot Pr_*(Q^{\varphi})}{Pr_*(\mathcal{X})} \quad (1)$$

Eq. (1) models the query posterior in terms of three components: (1) the semantic context prior $Pr_*(\mathcal{X})$, (2) the query prior $Pr_*(Q^{\varphi})$, and (3) the semantic context posterior $Pr_*(\mathcal{X} \mid Q^{\varphi})$. We proceed to analyze these components.

4.2.1. Semantic context prior

The semantic context prior $Pr_*(\mathcal{X})$ denotes the probability that any set of example tuples of size $|E|$ exhibits the semantic contexts $\mathcal{X}$. This probability is not easy to compute analytically, as it involves computing a marginal over a potentially infinite set of candidate queries. In this work, we model the semantic context prior as proportional to the *selectivity* $\psi(\Phi)$ of $\Phi = \{\phi_1, \phi_2, \dots\}$, where $\phi_i \in \Phi$ is a filter that encodes context $x_i \in \mathcal{X}$:

$$Pr_*(\mathcal{X}) \propto \psi(\Phi) \quad (2)$$

Selectivity $\psi(\phi)$. Selectivity of filter ϕ denotes the portion of tuples from the result of the base query Q^* that satisfy ϕ:

$$\psi(\phi) = \frac{|Q^{\{\phi\}}(D)|}{|Q^*(D)|}$$

Similarly, for a set of filters Φ, $\psi(\Phi) = \frac{|Q^{\Phi}(D)|}{|Q^*(D)|}$. Intuitively, a selectivity value close to 1 means that the filter is not very selective and most tuples satisfy the filter; selectivity value close to 0 denotes that the filter is highly selective and rejects most of the tuples. For example, in Fig. 7, $\phi_{\langle \texttt{gender}, \texttt{Male}, \perp\rangle}$ is more selective than $\phi_{\langle \texttt{age}, [50,90], \perp\rangle}$, with selectivities $\frac{1}{2}$ and $\frac{5}{6}$, respectively. Selectivity captures the rarity of a semantic context: uncommon contexts are present in fewer tuples, and, thus, appear in the output of fewer queries. Intuitively, a rare context has lower prior probability of being observed, which supports the assumption of Eq. (2).

4.2.2. Query prior

The query prior $Pr_*(Q^{\varphi})$ denotes the probability that Q^{φ} is the intended query, prior to observing the example tuples. We model the query prior as the joint probability of all filter events $\widetilde{\phi}$, where $\phi \in \Phi$. By further assuming filter independence,[5] we reduce the query prior to a product of probabilities of filter events:

$$Pr_*(Q^{\varphi}) = Pr_*(\textstyle\bigcap_{\phi \in \Phi} \widetilde{\phi}) = \prod_{\phi \in \Phi} Pr_*(\widetilde{\phi}) \quad (3)$$

The *filter event prior* $Pr_*(\widetilde{\phi})$ denotes the prior probability that filter ϕ is included in (if $\widetilde{\phi} = \phi$), or excluded from (if $\widetilde{\phi} = \bar{\phi}$), the intended query. We compute $Pr_*(\widetilde{\phi})$ for each filter as follows:

$$Pr_*(\phi) = \rho \cdot \delta(\phi) \cdot \alpha(\phi) \cdot \lambda(\phi) \quad \text{and} \quad Pr_*(\bar{\phi}) = 1 - Pr_*(\phi)$$

Here, ρ is a base prior parameter, common across all filters, and represents the default value for the prior. The other factors (δ, α, and λ) reduce the prior, depending on characteristics of each filter. We describe these parameters next.

Domain selectivity impact $\delta(\phi)$. Intuitively, a filter that covers a large range of values in an attribute's domain is unlikely to be part of the intended query. For example, if a user is interested in actors of a certain age group, that age group is more likely to be narrow ($\phi_{\langle \texttt{age}, [41,45], \perp\rangle}$) than broad ($\phi_{\langle \texttt{age}, [41,90], \perp\rangle}$). We penalize broad filters with the parameter $\delta \in (0, 1]$. The value $\delta(\phi)$ is 1 for filters that do not exceed a predefined ratio in the coverage of their domain, and decreases for filters that exceed this threshold. We use the notion of *domain coverage* of a filter $\phi_{\langle A, V, \theta\rangle}$ to denote the fraction of values of A's domain that V covers. For example, for attribute age, suppose that the domain consists of values in the range $[1, 100]$, then the filter $\phi_{\langle \texttt{age}, [41,90], \perp\rangle}$ has 50% domain coverage and the filter $\phi_{\langle \texttt{age}, [41,45], \perp\rangle}$ has 5% domain coverage. We use a threshold $\eta > 0$ to specify how much domain coverage does not reduce the domain selectivity impact δ. Once the threshold is exceeded, then δ decreases as domain coverage increases. We use another parameter $\gamma \geq 0$ that states how strongly we want to penalize a filter for having large domain coverage. The value of $\gamma = 0$ implies that we do not penalize at all, i.e., all filters will have $\delta(\phi) = 1$. As γ increases, we reduce δ more for larger domain coverages. Thus: $\delta(\phi_{\langle A, V, \theta\rangle}) = \frac{1}{\max(1, \frac{\texttt{domainCoverage}(V)}{\eta})^{\gamma}}$

Association strength impact $\alpha(\phi)$. Intuitively, a derived filter with low association strength is unlikely to appear in the intended query, as the filter denotes a weak association with the relevant entities. For example, $\phi_{\langle \texttt{genre}, \texttt{Comedy}, 1\rangle}$ is less likely than $\phi_{\langle \texttt{genre}, \texttt{Comedy}, 30\rangle}$ to represent a query intent. We label filters with θ lower than a threshold τ_{α} as insignificant, and set $\alpha(\phi) = 0$. All other filters, including basic filters, have $\alpha(\phi) = 1$.

Outlier impact $\lambda(\phi)$. While $\alpha(\phi)$ characterizes the impact of association strength on a filter individually, $\lambda(\phi)$ characterizes its impact in

[5] Reasoning about database queries commonly assumes independence across selection predicates, which filters represent, although it may not hold in general.

Case A

ϕ_1	$\phi\langle$genre, Comedy, 30$\rangle$
ϕ_2	$\phi\langle$genre, SciFi, 25$\rangle$
ϕ_3	$\phi\langle$genre, Drama, 3$\rangle$
ϕ_4	$\phi\langle$genre, Action, 2$\rangle$
ϕ_5	$\phi\langle$genre, Thriller, 1$\rangle$

Case B

ϕ_1	$\phi\langle$genre, Comedy, 12$\rangle$
ϕ_2	$\phi\langle$genre, SciFi, 10$\rangle$
ϕ_3	$\phi\langle$genre, Drama, 10$\rangle$
ϕ_4	$\phi\langle$genre, Action, 9$\rangle$
ϕ_5	$\phi\langle$genre, Thriller, 9$\rangle$

Fig. 8. Top two filters of Case A are interesting, whereas no filter is interesting in Case B.

consideration with other derived filters over the same attribute. Fig. 8 demonstrates two cases of derived filters on the same attribute (genre), corresponding to two different sets of example tuples. In Case A, ϕ_1 and ϕ_2 are more significant than the other filters of the same family (higher association strength). Intuitively, this corresponds to the intent to retrieve actors who appeared in mostly Comedy and SciFi movies. In contrast, Case B does not have filters that stand out, as all have similar association strengths: The actors in this example set are not strongly associated with particular genres, and, thus, intuitively, this family of filters is not relevant to the query intent.

We model the outlier impact $\lambda(\phi)$ of a filter using the *skewness* of the distribution of the association strengths within the family of derived filters sharing the same attribute. Our assumption is that highly skewed, heavy-tailed distributions (Case A) are likely to contain the significant (intended) filters as *outliers*. We set $\lambda(\phi) = 1$ for a derived filter whose association strength is an outlier in the association strength distribution of filters of the same family. We also set $\lambda(\phi) = 1$ for basic filters. All other filters get $\lambda(\phi) = 0$. Towards computing outlier impact of a filter $\phi_{\langle A,V,\theta\rangle}$, we first compute skewness of the association strength distribution Θ_A within the family of derived filters involving attribute A; and then check whether θ is an outlier among them. We compute sample skewness of $\Theta_A = (a_1, a_2, \ldots, a_n)$, with sample mean $\bar{a}$ and sample standard deviation s, using the standard formula:

$$\texttt{skewness}(\Theta_A) = \frac{n\sum_{i=1}^{n}(a_i - \bar{a})^3}{s^3(n-1)(n-2)}$$

A distribution is skewed if its skewness exceeds a threshold τ_s. For outlier detection, we use the mean and standard deviation method. For sample mean $\bar{a}$, sample standard deviation s, and a constant $k \geq 2$, a_i is an outlier if $(a_i - \bar{a}) > ks$. For $n < 3$, skewness is not defined and we assume all elements to be outliers. We compute outlier impact $\lambda(\phi_{\langle A,V,\theta\rangle})$:

$$\lambda(\phi_{\langle A,V,\theta\rangle}) = \begin{cases} 1 & \theta = \bot \vee (\texttt{skewness}(\Theta_A) > \tau_s \wedge \texttt{outlier}(\theta)) \\ 0 & \text{otherwise} \end{cases}$$

4.2.3. Semantic context posterior

The semantic context posterior $Pr_*(\mathcal{X} \mid Q^\varphi)$ is the probability that a set of example tuples of size $|E|$, sampled from the output of a particular query Q^φ, exhibits the set of semantic contexts $\mathcal{X}$:

$$Pr_*(\mathcal{X} \mid Q^\varphi) = Pr_*(x_1, x_2, \ldots, x_n \mid Q^\varphi)$$

Two semantic contexts $x_i, x_j \in \mathcal{X}$ are conditionally independent, given Q^φ. Therefore:

$$Pr_*(\mathcal{X} \mid Q^\varphi) = \prod_{i=1}^{n} Pr_*(x_i \mid Q^\varphi) = \prod_{i=1}^{n} Pr_*(x_i \mid \widetilde{\phi}_1, \widetilde{\phi}_2, \ldots)$$

Recall that ϕ_i encodes the semantic context x_i (Section 4.1). We assume that x_i is conditionally independent of any $\widetilde{\phi}_j, i \neq j$, given $\widetilde{\phi}_i$ (this always holds for $\widetilde{\phi}_i = \phi_i$):

$$Pr_*(\mathcal{X} \mid Q^\varphi) = \prod_{i=1}^{n} Pr_*(x_i \mid \widetilde{\phi}_i) \tag{4}$$

For each x_i, we compute $Pr_*(x_i \mid \widetilde{\phi}_i)$ based on the state of the filter event ($\widetilde{\phi}_i = \phi_i$ or $\widetilde{\phi}_i = \bar{\phi}_i$):

$Pr_*(x_i \mid \phi_i)$: By definition, all tuples in $Q^{\{\phi_i\}}(D)$ exhibit the property of x_i. Hence, $Pr_*(x_i \mid \phi_i) = 1$.

$Pr_*(x_i \mid \bar{\phi}_i)$: This is the probability that a set of $|E|$ tuples drawn uniformly at random from $Q^*(D)$ (ϕ_i is not applied to the base query) exhibits the context x_i. The portion of tuples in $Q^*(D)$ that exhibit the property of x_i is the selectivity $\psi(\phi_i)$. Therefore, $Pr_*(x_i \mid \bar{\phi}_i) \approx \psi(\phi_i)^{|E|}$.

Using Eqs. (1)–(4), we derive the final form of the query posterior (where K is a normalization constant):

$$\begin{aligned} Pr_*(Q^\varphi \mid E) &= \frac{K}{\psi(\boldsymbol{\Phi})} \prod_{\phi_i \in \boldsymbol{\Phi}} \left(Pr_*(\widetilde{\phi}_i) \cdot Pr_*(x_i \mid \widetilde{\phi}_i) \right) \\ &= \frac{K}{\psi(\boldsymbol{\Phi})} \prod_{\phi_i \in \varphi} \left(Pr_*(\phi_i) \cdot Pr_*(x_i \mid \phi_i) \right) \prod_{\phi_i \notin \varphi} \left(Pr_*(\bar{\phi}_i) \cdot Pr_*(x_i \mid \bar{\phi}_i) \right) \end{aligned} \tag{5}$$

4.3. Generalization

So far, our analysis focused on a fixed base query. Given an SPJ query Q^φ, the underlying base query Q^* is deterministic, i.e., $Pr(Q^* \mid Q^\varphi) = 1$. Hence:

$$\begin{aligned} Pr(Q^\varphi \mid E) &= Pr(Q^\varphi, Q^* \mid E) \\ &= Pr(Q^\varphi \mid Q^*, E) \cdot Pr(Q^* \mid E) \\ &= Pr_*(Q^\varphi \mid E) \cdot Pr(Q^* \mid E) \end{aligned}$$

We assume $Pr(Q^* \mid E)$ to be equal for all valid base queries, where $Q^*(D) \supseteq E$. Then we use $Pr_*(Q^\varphi \mid E)$ to find the query Q that maximizes the query posterior $Pr(Q \mid E)$.

5. Offline abduction preparation

In this section, we discuss system considerations to perform query intent discovery *efficiently*. SQuID employs an offline module that performs several precomputation steps to make the database *abduction-ready*. The abduction-ready database (αDB) augments the original database with derived relations that store associations across entities and precomputes semantic property statistics. Deriving this information is relatively straightforward; the contributions of this section lie in the design of the αDB, the information it maintains, and its role in supporting efficient query intent discovery. We describe the three major functions of the αDB.

5.1. Entity lookup

SQuID's goal is to discover query intent based on the user-provided examples. To do that, it first needs to determine which entities in the database correspond to the examples. SQuID uses a *global inverted column index* [7], built over all text attributes and stored in the αDB, to perform fast lookups, matching the provided examples to database entities.

5.2. Semantic property discovery

To reason about intent, SQuID first needs to determine what makes the examples similar. SQuID looks for semantic properties within

entity relations (e.g., `gender` appears in table `person`), other relations (e.g., `genre` appears in a separate table joining with `movie` through a key-foreign-key constraint), and other entities, (e.g., the number of movies of a particular `genre` that a `person` has appeared in). The αDB precomputes and stores such *derived relations* (e.g., `persontogenre`), as these frequently involve several joins and aggregations and performing them at runtime is prohibitive. E.g., SQuID computes the `persontogenre` relation (Fig. 5) and stores it in the αDB with the SQL query below:

```
Q6: CREATE TABLE persontogenre as
    (SELECT person_id, genre_id, count(*) AS count
     FROM castinfo, movietogenre
     WHERE castinfo.movie_id = movietogenre.movie_id
     GROUP BY person_id, genre_id)
```

For the αDB construction, SQuID only relies on very basic information to understand the data organization: (1) the database schema, including the specification of primary- and foreign-key constraints, and (2) additional meta-data, which can be provided once by a database administrator, that specify which tables describe entities (e.g., `person`, `movie`), and which tables and attributes describe direct properties of entities (e.g., `genre`, `age`). SQuID then automatically discovers fact tables, which associate entities and properties, by exploiting the key-foreign-key relationships. SQuID also automatically discovers derived properties up to a certain predefined depth, using paths in the schema graph that connect entities to properties. Since the number of possible values for semantic properties is typically very small and remains constant as entities grow, the αDB grows linearly with the data size. In our implementation, we restrict the derived property discovery to the depth of two fact tables (e.g., SQuID derives `persontogenre` through `castinfo` and `movietogenre`). SQuID can support deeper associations, but those are rare in practice. SQuID assumes that different entity types appear in different relations, which is the case in many commonly used schema types, such as star, galaxy, and fact-constellation schemas. SQuID can perform inference in a denormalized setting, but would not be able to produce and reason about derived properties in those cases.

5.3. Smart selectivity computation

For basic filters involving categorical values, SQuID stores the selectivity for each value. However, for numeric ranges, the number of possible filters can grow quadratically with the number of possible values. SQuID avoids wasted computation and space by only precomputing selectivities $\psi(\phi_{\langle A,[min_{V_A},v],\perp\rangle})$ for all $v \in V_A$, where V_A is the set of values of attribute A in the corresponding relation, and min_{V_A} is the minimum value in V_A. The αDB can derive the selectivity of a filter with any value range as:

$$\psi(\phi_{\langle A,[l,h],\perp\rangle}) = \psi(\phi_{\langle A,[min_{V_A},h],\perp\rangle}) - \psi(\phi_{\langle A,[min_{V_A},l],\perp\rangle})$$

In case of derived semantic properties, SQuID precomputes selectivities $\psi(\phi_{\langle A,v,\theta\rangle})$ for all $v \in V_A, \theta \in \Theta_{A,v}$, where $\Theta_{A,v}$ is the set of values of association strength for the property "$A = v$".

6. Query intent discovery

During normal operation, SQuID receives example tuples from a user, consults the αDB, and infers the most likely query intent (Definition 2.1). In this section, we describe how SQuID resolves ambiguity in the provided examples, how it derives their semantic context, and how it finally abduces the intended query.

6.1. Entity and context discovery

SQuID's probabilistic abduction model (Section 4) relies on the set of semantic contexts $\mathcal{X}$ and determines which of these contexts are intended vs coincidental, by the inclusion or exclusion of the corresponding filters in the inferred query. To derive the set of semantic contexts from the examples, SQuID first needs to identify the entities in the αDB that correspond to the provided examples.

6.1.1. Entity disambiguation

User-provided examples are not complete tuples, but often single-column values that correspond to an entity. As a result, there may be ambiguity that SQuID needs to resolve. For example, suppose the user provides the examples: {`Titanic`, `Pulp Fiction`, `The Matrix`}. SQuID consults the precomputed inverted column index to identify the attributes (`movie.title`) that contain all the example values, and classifies the corresponding entity (`movie`) as a potential match. However, while the dataset contains unique entries for `Pulp Fiction (1994)` and `The Matrix (1999)`, there are 4 possible mappings for `Titanic`: (1) a 1915 Italian film, (2) a 1943 German film, (3) a 1953 film by Jean Negulesco, and (4) the 1997 blockbuster film by James Cameron.

While there is a rich line of work dedicated for entity disambiguation [27], they do not trivially extend to our case. Entity disambiguation typically works on a pair of entities in isolation. In contrast, here, we can leverage the accompanying example tuples as a useful context. The key insight for resolving ambiguities here is that the provided examples are more likely to be alike. SQuID selects the entity mappings that maximize the semantic similarities across the examples. Therefore, based on the year and country information, it determines that `Titanic` corresponds to the 1997 film, as it is most similar to the other two (unambiguous) entities. In case of derived properties, e.g., nationality of actors appearing in a film, SQuID aims to increase the association strength (e.g., the number of such actors). Since the number of examples are typically small, SQuID can determine the right mappings by considering all combinations.

6.1.2. Semantic context discovery

Once SQuID identifies the right entities, it then explores all the semantic properties stored in the αDB that match these entities (e.g., `year`, `genre`, etc.). Since the αDB precomputes and stores the derived properties, SQuID can produce all the relevant properties using queries with at most one join. For each property, SQuID produces semantic contexts as follows:

Basic property on categorical attribute. If all examples in E contain value v for the property of attribute A, SQuID produces the semantic context $(\langle A, v, \perp\rangle, |E|)$. For example, a user provides three movies: `Dunkirk`, `Logan`, and `Taken`. The attribute `genre` corresponds to a basic property for movies, and all these movies share the values, `Action` and `Thriller`, for this property. SQuID generates two semantic contexts: $(\langle \texttt{genre}, \texttt{Action}, \perp\rangle, 3)$ and $(\langle \texttt{genre}, \texttt{Thriller}, \perp\rangle, 3)$.

Basic property on numerical attribute. If v_{min} and v_{max} are the minimum and maximum values, respectively, that the examples in E demonstrate for the property of attribute A, SQuID creates a semantic context on the range $[v_{min}, v_{max}]$: $(\langle A, [v_{min}, v_{max}], \perp\rangle, |E|)$. For example, if E contains three persons with ages 45, 50, and 52, SQuID will produce the context $(\langle \text{age}, [45, 52], \perp\rangle, 3)$.

Derived property. If all examples in E contain value v for the derived property of attribute A, SQuID produces the context $(\langle A, v, \theta_{min}\rangle, |E|)$, where θ_{min} is the minimum association strength for the value v among all examples. For example, if E contains two persons who appeared in 3 and 5 `Comedy` movies, SQuID will produce $(\langle \texttt{genre}, \texttt{Comedy}, 3\rangle, 2)$.

6.2. Query abduction

SQuID starts abduction by constructing a base query that captures the structure of the example tuples. Once it identifies the entity and attribute that matches the examples (e.g., `person.name`), it forms the minimal PJ query (e.g., `SELECT name FROM person`). It then iterates

Algorithm 1: QueryAbduction (E, Q^*, Φ)

Input: set of entities E, base query Q^*, set of minimal valid filters Φ
Output: Q^φ such that $Pr_*(Q^\varphi \mid E)$ is maximized

1 $\mathcal{X} = \{x_1, x_2, ...\}$ // semantic contexts in E
2 $\varphi = \emptyset$
3 **foreach** $\phi_i \in \Phi$ **do**
4 $\quad \texttt{include}_{\phi_i} = Pr_*(\phi_i) Pr_*(x_i \mid \phi_i)$ // from Equation (5)
5 $\quad \texttt{exclude}_{\phi_i} = Pr_*(\bar{\phi}_i) Pr_*(x_i \mid \bar{\phi}_i)$ // from Equation (5)
6 $\quad$ **if** $\texttt{include}_{\phi_i} > \texttt{exclude}_{\phi_i}$ **then**
7 $\quad\quad \varphi = \varphi \cup \{\phi_i\}$
8 **return** Q^φ

through the discovered semantic contexts and appends the corresponding relations to the FROM clause and the appropriate key-foreign-key join conditions in the WHERE clause. Since the αDB precomputes and stores the derived relations, each semantic context will add at most one relation to the query.

The number of candidate base queries is typically very small. For each base query Q^*, SQuID abduces the best set of filters $\varphi \subseteq \Phi$ to construct SPJ query Q^φ, by augmenting the WHERE clause of Q^* with the corresponding selection predicates. SQuID also removes from Q^φ any joins that are not relevant to the selected filters φ. While the number of candidate SPJ queries grows exponentially in the number of minimum valid filters ($2^{|\Phi|}$), we prove that we can make decisions on including or excluding each filter independently. Algorithm 1 iterates over the set of minimal valid filters Φ and decides to include a filter only if its addition to the query increases the query posterior (lines 6–7). Our abduction algorithm has $O(|\Phi|)$ time complexity and is guaranteed to produce the query Q^φ that maximizes the query posterior.

Theorem 1. *Given a base query Q^*, a set of examples E, and a set of minimal valid filters Φ, Algorithm 1 returns the query Q^φ, where $\varphi \subseteq \Phi$, such that $Pr_*(Q^\varphi \mid E)$ is maximized.*

Proof. We prove Theorem 1 by contradiction. Suppose that φ is the optimal set of filters, i.e., Q^φ is the most likely query. Additionally, suppose that φ is the minimal set of filters for obtaining such optimality, i.e., $\nexists \varphi'$ such that $|\varphi'| < \varphi \wedge Pr(Q^{\varphi'} \mid E) = Pr(Q^\varphi \mid E)$. Now suppose that, Algorithm 1 returns a sub-optimal query $Q^{\varphi'}$, i.e., $Pr(Q^{\varphi'} \mid E) < Pr(Q^\varphi \mid E)$. Since $Q^{\varphi'}$ is suboptimal, $\varphi' \neq \varphi$; therefore at least one of the following two cases must hold:

Case 1: $\exists \phi$ such that $\phi \in \varphi \wedge \phi \notin \varphi'$. Since Algorithm 1 did not include ϕ, it must be the case that $\texttt{include}_\phi \leq \texttt{exclude}_\phi$. Therefore, we can exclude ϕ from φ to obtain $\varphi - \{\phi\}$ and according to Eq. (5), $Pr(Q^{\varphi - \{\phi\}} \mid E) \geq Pr(Q^\varphi \mid E)$ which contradicts with our assumption about the optimality and minimality of φ.

Case 2: $\exists \phi$ such that $\phi \notin \varphi \wedge \phi \in \varphi'$. Since Algorithm 1 included ϕ, it must be the case that $\texttt{include}_\phi > \texttt{exclude}_\phi$. Therefore, we can add ϕ to φ to obtain $\varphi \cup \{\phi\}$ and according to Eq. (5), $Pr(Q^{\varphi \cup \{\phi\}} \mid E) > Pr(Q^\varphi \mid E)$ which again contradicts with our assumption about the optimality of φ.

Hence, $Q^{\varphi'}$ cannot be suboptimal and this implies that Algorithm 1 returns the most likely query. □

In a special case where $\texttt{include}_\phi = \texttt{exclude}_\phi$, Algorithm 1 drops the filter using Occam's razor principle to keep the query as simple as possible. However, this does not return any query that is strictly less likely than the best query.

7. Experiments

In this section, we present an extensive experimental evaluation of SQuID over three real-world datasets, with a total of 41 benchmark queries of varying complexities. Our evaluation shows that SQuID is scalable and effective, even with a small number of example tuples. Our evaluation extends to qualitative case studies over real-world, user-generated examples, which demonstrate that SQuID succeeds in inferring the query intent of real-world users. We further demonstrate that when used as a query reverse engineering system in a closed-world setting, SQuID outperforms the state of the art. Finally, we show that SQuID is superior to semi-supervised PU-learning in terms of both efficiency and effectiveness.

7.1. Experimental setup

We implemented SQuID in Java and all experiments were run on a 12 × 2.66 GHz machine with 16 GB RAM running CentOS 6.9 with PostgreSQL 9.6.6.

Datasets and Benchmark Queries. Our evaluation includes three real-world datasets and a total of 41 benchmark queries, designed to cover a broad range of intents and query structures. We summarize the datasets and queries below and provide detailed description in Appendix.

IMDb (633 MB): The dataset contains 15 relations with information on movies, cast members, film studios, etc. We designed a set of 16 benchmark queries ranging the number of joins (1 to 8 relations), the number of selection predicates (0 to 7), and the result cardinality (12 to 2512 tuples).

DBLP (22 MB): We used a subset of the DBLP data [28], with 14 relations, and 16 years (2000–2015) of top 81 conference publications. We designed 5 queries ranging the number of joins (3 to 8 relations), the number of selection predicates (2 to 4), and the result cardinality (15 to 468 tuples).

Adult (4 MB): This is a single relation dataset containing census data of people and their income brackets. We generated 20 queries, randomizing the attributes and predicate values, ranging the number of selection predicates (2 to 7) and the result cardinality (8 to 1404 tuples).

Case Study Data. We retrieved several public lists (sources are listed in Appendix) with human-generated examples, and identified the corresponding intent. For example, a user-created list of "115 funniest actors" reveals a query intent (funny actors), and provides us with real user examples (the names in the list). We used this method to design 3 case studies: funny actors (IMDb), 2000s Sci-Fi movies (IMDb), and prolific database researchers (DBLP).

Metrics. We report query discovery time as a metric of efficiency. We measure effectiveness using precision, recall, and f-score. If Q is the intended query, and Q' is the query inferred by SQuID, precision is computed as $\frac{Q'(D) \cap Q(D)}{Q'(D)}$ and recall as $\frac{Q'(D) \cap Q(D)}{Q(D)}$; f-score is their harmonic mean. We also report the total number of predicates in the produced queries and compare them with the actual intended queries.

Comparisons. To the best of our knowledge, existing QBE techniques do not produce SPJ queries without (1) a large number of examples, or (2) additional information, such as provenance. For this reason, we cannot meaningfully compare SQuID with those approaches. Removing the open-world requirement, SQuID is most similar to the QRE system TALOS [20] with respect to expressiveness and capabilities (Fig. 3). We compare the two systems for query reverse engineering tasks in Section 7.5. We also compare SQuID against PU-learning methods [21] in Section 7.6.

7.2. Scalability

In our first set of experiments, we examine the scalability of SQuID against increasing number of examples and varied dataset sizes. Fig. 9(a) displays the abduction time for the IMDb and DBLP datasets as the

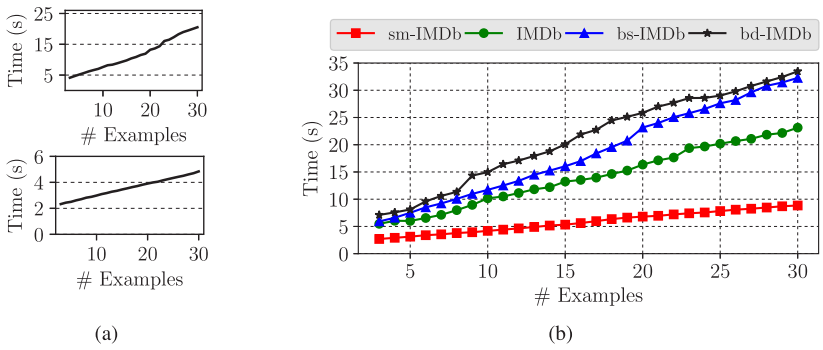


Fig. 9. Average abduction time over the benchmark queries in (a) IMDb (top), DBLP (bottom), and (b) 4 versions of the IMDb dataset in different sizes.

number of provided examples increases, averaged over all benchmark queries in each dataset. Since SQuID retrieves semantic properties and computes context for each example, the runtime increases linearly with the number of examples, which is what we observe.

Fig. 9(b) extends this experiment to datasets of varied sizes. We generate three alternative versions of the IMDb dataset: (1) sm-IMDb (75 MB), a downsized version that keeps 10% of the original data; (2) bs-IMDb (1330 MB), doubles the entities of the original dataset and creates associations among the duplicate entities (`person` and `movie`) by replicating their original associations; (3) bd-IMDb (1926 MB), is the same as bs-IMDb but also introduces associations between the original entities and the duplicates, creating denser connections.[6] SQuID's runtime increases for all datasets with the number of examples, and, predictably, larger datasets face longer abduction times. Query abduction involves point queries to retrieve semantic properties of the entities, using B-tree indexes. As the data size increases, the runtime of these queries grows logarithmically. SQuID is slower on bd-IMDb than on bs-IMDb: both datasets include the same entities, but bd-IMDb has denser associations, which results in additional derived semantic properties.

7.3. Abduction accuracy

Intuitively, with a larger number of examples, abduction accuracy should increase: SQuID has access to more samples of the query output, and can more easily distinguish coincidental from intended similarities. Fig. 10 confirms this intuition, and precision, recall, and f-score increase, often very quickly, with the number of examples for most of our benchmark queries. We discuss here a few particular queries.

`IQ4` & `IQ11`: These queries include a statistically common property (USA movies), and SQuID needs more examples to confirm that the property is indeed intended, not coincidental; hence, the precision converges more slowly.

`IQ6`: In many movies where Clint Eastwood was a director, he was also an actor. SQuID needs to observe sufficient examples to discover that the property `role:Actor` is not intended, so recall converges more slowly.

`IQ10`: SQuID performs poorly for this query. The query looks for actors appearing in more than 10 *Russian* movies that were released *after 2010*. While SQuID discovers the derived properties "more than 10 Russian movies" and "more than 10 movies released after 2010", it cannot compound the two into "more than 10 Russian movies released after 2010". This query is simply outside of SQuID's search space, and SQuID produces a query with more general predicates than intended, which is why precision drops.

`IQ3`: The query is looking for actresses who are Canadian and were born after 1970. SQuID successfully discovers the properties `gender: Female`, `country: Canada`, and `birth year` $\geq$ `1970`; however, it fails to capture the property of "being an actress", corresponding to having appeared in at least 1 film. The reason is that SQuID is programmed to ignore weak associations (a person associated with only 1 movie). This behavior can be fixed by adjusting the association strength parameter to allow for weaker associations.

Generally, missing an intended property results in low precision and picking an unintended property results in low recall. When SQuID makes both mistakes at the same time, we see low precision and low recall.

7.3.1. Execution time

While the accuracy results demonstrate that the abduced queries are semantically close to the intended queries, SQuID could be deriving a query that is semantically close, but more complex and costly to compute. In Figs. 11(a) and 11(b) we graph the average runtime of the abduced queries and the actual benchmark queries. We observe that in most cases the abduced queries and the corresponding benchmarks are similar in execution time. Frequently, the abduced queries are faster because they take advantage of the precomputed relations in the αDB. In few cases (`IQ1`, `IQ5`, and `IQ7`) SQuID discovered additional properties that, while not specified by the original query, are inherent in all intended entities. For example, in `IQ5`, all movies with `Tom Cruise` and `Nicole Kidman` are also English language movies and released between 1990 and 2014.

7.3.2. Effect of entity disambiguation

Finally, we found that entity disambiguation never hurts abduction accuracy, and may significantly improve it. Fig. 12 displays the impact of disambiguation for five IMDb benchmark queries, where disambiguation significantly improves the f-score.

7.4. Qualitative case studies

We now present qualitative results on the performance of SQuID, through a simulated user study. We designed 3 case studies, by constructing queries and examples from human-generated, publicly available lists.

Funny actors (IMDb). We created a list of names of 211 "funny actors", collected from human-created public lists and Google Knowledge Graph (sources are in Appendix), and used these names as examples of the query intent "funny actors". Fig. 13(a) demonstrates the accuracy of the abduced query over a varying number of examples. Each data point is an average across 10 different random samples of example sets of the corresponding size. For this experiment, we tuned SQuID to normalize the association strength, which means that the relevant predicate would consider the fraction of movies in an actor's portfolio classified as comedies, rather than the absolute number.

2000S sci-fi movies (IMDb). We used a user-created list of 165 Sci-Fi movies released in 2000s as examples of the query intent "2000s

[6] Details of the data generation process are in Appendix.

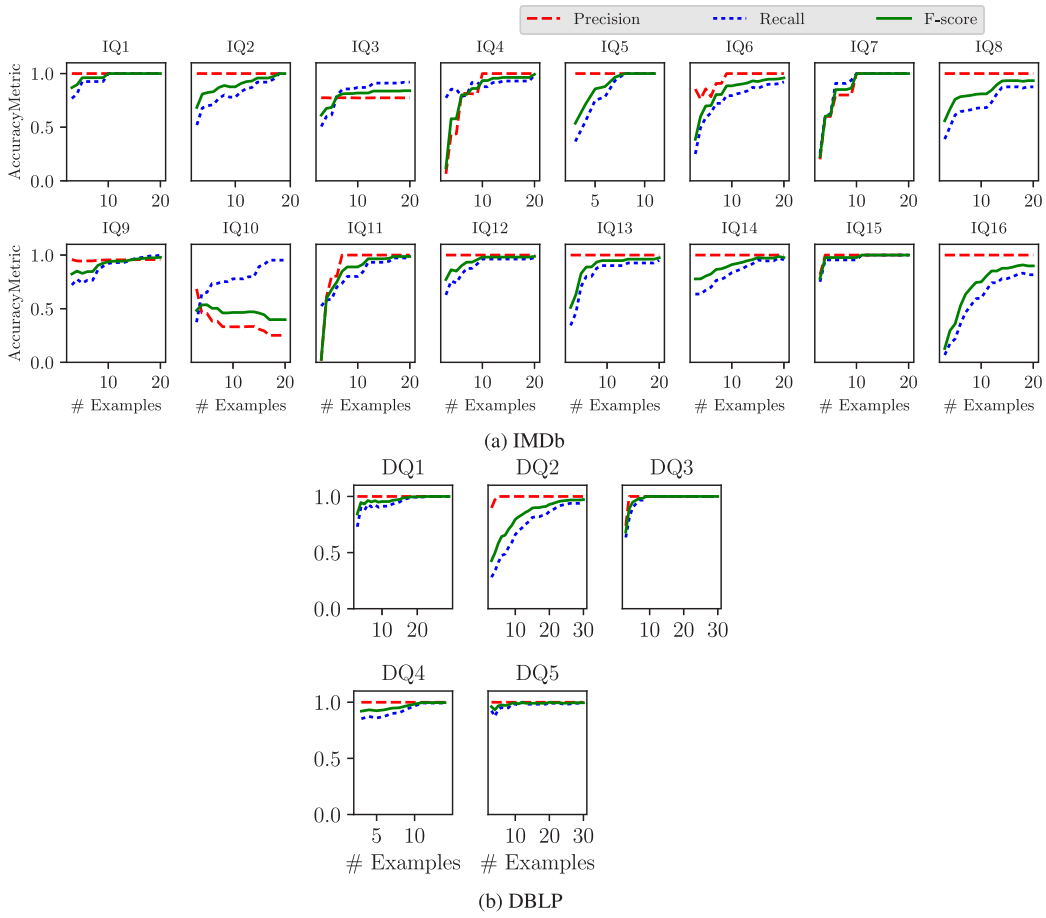


Fig. 10. SQuID achieves high accuracy with few examples (typically ~5) in most benchmark queries.

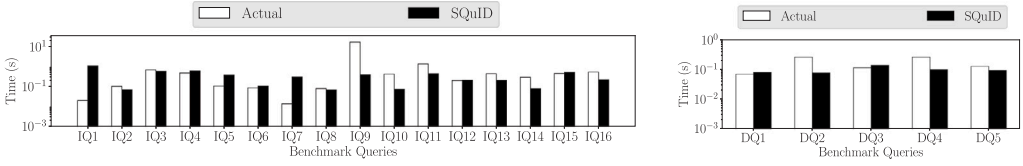


Fig. 11. SQuID rarely produces queries that are slower than the original with respect to query runtime.

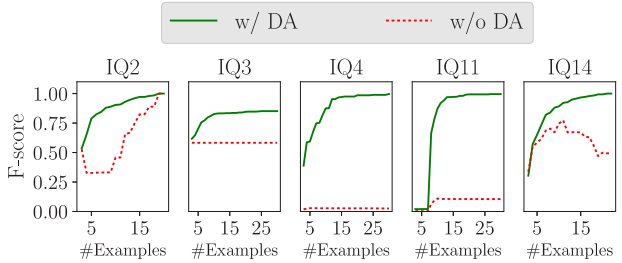


Fig. 12. Effect of disambiguation on IMDb.

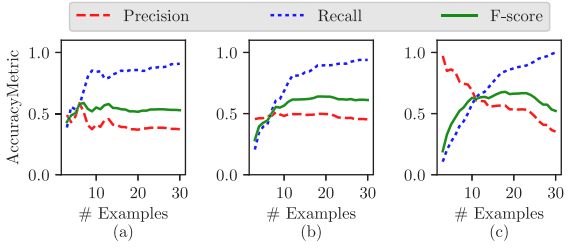


Fig. 13. Precision, recall, and f-score for (a) Funny actors (b) 2000s Sci-Fi movies (c) Prolific DB researchers.

Sci-Fi movies". Fig. 13(b) displays the accuracy of the abduced query, averaged across 10 runs for each example set size.

Prolific database researchers (DBLP). We collected a list of database researchers who served as chairs, group leaders, or program committee members in SIGMOD 2011–2015 and selected the top 30 most prolific. Fig. 13(c) displays the accuracy of the abduced query averaged, across 10 runs for each example set size.

7.4.1. Analysis

In our case studies there is no (reasonable) SQL query that models the intent well and produces an output that exactly matches our lists.

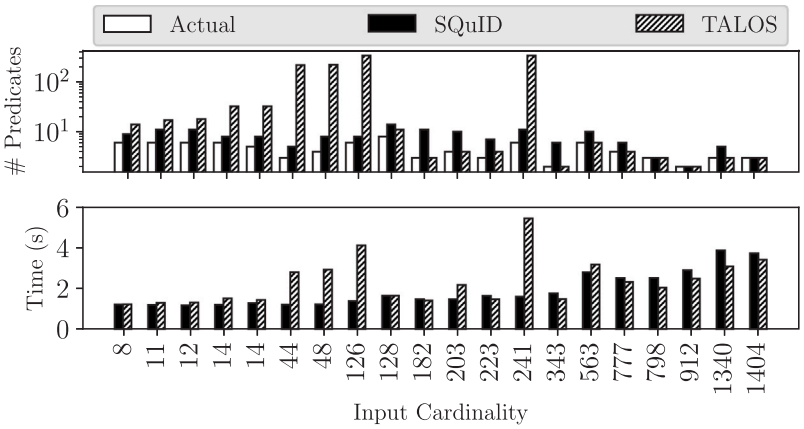
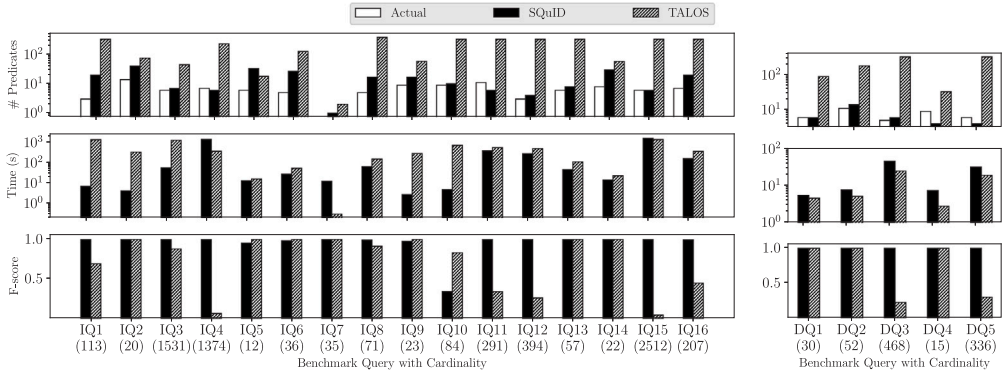


Fig. 14. Both systems achieve perfect f-score on the Adult dataset (not shown). SQuID produces significantly smaller queries, often by orders of magnitude, and is often much faster.



(a) IMDb (b) DBLP

Fig. 15. SQuID produces queries with significantly fewer predicates than TALOS and is more accurate on both IMDb and DBLP. SQuID is almost always faster on IMDb, but TALOS is faster on DBLP.

Public lists have biases, such as not including less well-known entities even if these match the intent. To counter this bias, we use popularity masks (derived from public lists) to filter the examples and the abduced query outputs Appendix. In our prolific researchers use case, some well-known and prolific researchers may happen to not serve in service roles frequently, or their commitments may be in venues we did not sample. Therefore, it is not possible to achieve high precision, as the data is bound to contain and retrieve entities that do not appear on the lists, even if the query is a good match for the intent. For this reason, our precision numbers in the case studies are low. However our recall rises quickly with enough examples, which indicates that the abduced queries converge to the correct intent.

7.5. Query reverse engineering

We present an experimental comparison of SQuID with TALOS [20], a state-of-the-art query reverse engineering (QRE) system. We picked TALOS because other related methods either focus on more restricted query classes [29,30] or do not scale to data sizes large enough for this evaluation [15,16] (Fig. 3). Unlike SQuID, QRE systems operate in a closed-world setting, assuming that the provided examples comprise the entire query output. In the closed-world setting, SQuID is handicapped against a dedicated QRE system, as it does not take advantage of the closed-world constraint.

For this evaluation under the QRE setting, we use the IMDb and DBLP datasets, as well as the Adult dataset, on which TALOS was shown to perform well [20]. For each dataset, we provided the entire output of the benchmark queries as input to SQuID and TALOS. Since there is no need to drop coincidental filters for query reverse engineering, we set the parameters so that SQuID behaves optimistically (e.g., high filter prior, low association strength threshold, etc.). We adopt the notion of *instance equivalent query* (IEQ) from the QRE literature [20] to express that two queries produce the same set of results on a particular database instance. A QRE task is successful if the system discovers an IEQ of the original query (f-score=1). For the IMDb dataset, SQuID was able to successfully reverse engineer 11 out of 16 benchmark queries. Additionally, in 4 cases where exact IEQs were not abduced, SQuID queries generated output with ≥0.98 f-score. SQuID failed only for `IQ10`, which is a query that falls outside the supported query family, as discussed in Section 7.3. For the DBLP and Adult datasets, SQuID successfully reverse-engineered all benchmark queries. We compare SQuID to TALOS on three metrics: number of predicates (including join and selection predicates), query discovery time, and f-score.

Adult. Both SQuID and TALOS achieved perfect f-score on the 20 benchmark queries. Fig. 14 compares the systems in terms of the number of predicates in the queries they produce (top) and query discovery time (bottom). SQuID almost always produces simpler queries, close in the number of predicates to the original query, while TALOS queries contain more than 100 predicates in 20% of the cases. SQuID is faster than TALOS when the input cardinality is low (~100 tuples), and becomes slower for the largest input sizes (>700 tuples). SQuID was not designed as a QRE system, and in practice, users rarely provide large example sets. SQuID's focus is on inferring simple queries that model the intent, rather than cover all examples with potentially complex and lengthy queries.

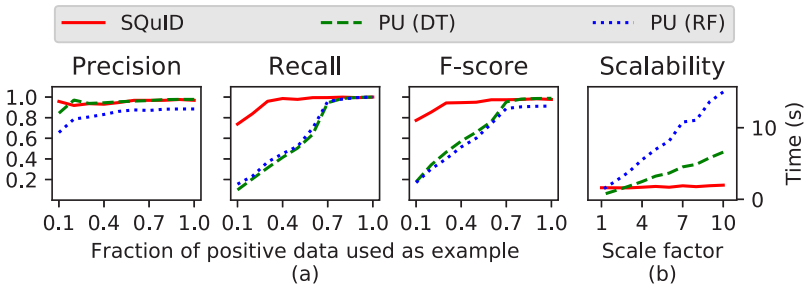


Fig. 16. (a) PU-learning needs a large fraction (>70%) of the query results (positive data) as examples to achieve accuracy comparable to SQuID. (b) The total required time for training and prediction in PU-learning increases linearly with the data size. In contrast, abduction time for SQuID increases logarithmically.

IMDb. Fig. 15(a) compares the two systems on the 16 benchmark queries over IMDb. SQuID produced better queries in almost all cases: in all cases, our abduced queries where significantly smaller, and our f-score is higher for most queries. SQuID was also faster than TALOS for most of the benchmark queries. We now delve deeper into some particular cases.

For `IQ1` (cast of `Pulp Fiction`), TALOS produces a query with f-score = 0.7. We attempted to provide guidance to TALOS through a system parameter that specifies which attributes to include in the selection predicates (which would give it an unfair advantage). TALOS first performs a full join among the participating relations (`person` and `castinfo`) and then performs classification on the denormalized table (with attributes person, movie, role). TALOS gives all rows referring to a cast member of `Pulp Fiction` a positive label (based on the examples), regardless of the movie that row refers to, and then builds a decision tree based on these incorrect labels. This is a limitation of TALOS, which SQuID overcomes by looking at the semantic similarities of the examples, rather than treating them simply as labels.

SQuID took more time than TALOS in `IQ4`, `IQ7`, and `IQ15`. The result sets of `IQ4` and `IQ15` are large (>1000), so this is expected. `IQ7` retrieves all movie genres, and, thus, does not require any selection predicate. As a decision tree approach, TALOS has the advantage here, as it stops at the root and does not need to traverse the tree. In contrast, SQuID retrieves all semantic properties of the example tuples only to discover that either there is nothing common among them, or the property is not significant. While SQuID takes longer, it still abduces the correct query. These are not representative of QBE scenarios, as users are unlikely to provide large number of example tuples or have very general intents (PJ queries without selection).

DBLP. Fig. 15(b) compares the two systems on the DBLP dataset. Here, SQuID successfully reverse engineered all five benchmark queries, but TALOS failed to reverse engineer two of them. TALOS also produced very complex queries, with 100 or more predicates for four of the cases. In contrast, SQuID's abductions were orders of magnitude smaller, on par with the original query. On this dataset, SQuID was slower than TALOS, but not by a lot.

7.6. Comparison with learning methods

Query intent discovery can be seen as a one-class classification problem, where the task is to identify the tuples that satisfy the desired intent. Positive and Unlabeled (PU) learning addresses this problem by learning a classifier from positive examples and unlabeled data in a semi-supervised setting. We compare SQuID against an established PU-learning method [21] on 20 benchmark queries over Adult. The setting of this experiment conforms with the technique's requirements [21]: the dataset comprises of a single relation and the examples are chosen uniformly at random from the positive data.

Fig. 16(a) compares the accuracy of SQuID and PU-learning using two different estimators, decision tree (DT) and random forest (RF). We observe that PU-learning needs a large fraction (>70%) of the query result to achieve f-score comparable to SQuID. PU-learning favors precision over recall, and the latter drops significantly when the number of examples is low. In contrast, SQuID achieves robust performance, even with few examples, because it can encode problem-specific assumptions (e.g., that there exists an underlying SQL query that models the intent, that some filters are more likely than other filters, etc.); this cannot be done in straightforward ways for machine learning methods.

To evaluate scalability, we replicated the Adult dataset, with a scale factor up to 10x. Fig. 16(b) shows that PU-learning becomes significantly slower than SQuID as the data size increases, whereas SQuID's runtime performance remains largely unchanged. This is due to the fact that, SQuID does not directly operate on the data outside of the examples (unlabeled data); rather, it relies on the αDB, which contains a highly compressed summary of the semantic property statistics (e.g., filter selectivities) of the data. In contrast, PU-learning builds a new classifier over all of the data for each query intent discovery task. We provide more discussion on the connections between SQuID and machine learning approaches in Section 10.

7.7. SQuID parameters

We list the four most important SQuID parameters in Fig. 17 along with brief description. We now discuss the impact of these parameters on SQuID and provide few empirical results.

ρ. The base filter prior parameter ρ defines SQuID's tendency towards including filters. Small ρ makes SQuID pessimistic about including a filter, and thus favors recall. In contrast, large ρ makes SQuID optimistic about including a filter, which favors precision. A low ρ helps in getting rid of coincidental filters, particularly with very few example tuples. However, with sufficient example tuples, coincidental filters eventually disappears, and the effect of ρ diminishes. Fig. 18(a) shows effect of varying the value of ρ for a few benchmark queries on the IMDb dataset. While a low ρ favors some queries (`IQ2`, `IQ16`), it causes accuracy degradation for some other queries (`IQ3`, `IQ4`, `IQ11`), where a high ρ works better. It is a tradeoff and we found empirically that moderate value of ρ (e.g., 0.1) works best on an average.

γ. The domain coverage penalty parameter γ specifies SQuID's leniency towards filters with large domain coverage. Low γ penalizes filters with large domain coverage less, and a high γ penalizes them more. Fig. 18(b) shows the effect of varying γ. Very low value for γ favors some queries (`IQ3`, `IQ4`, `IQ11`) but also causes accuracy degradation for some other queries (`IQ2`, `IQ16`), where a high γ works better. Like ρ, it is also a tradeoff, and empirically we found moderate values of γ (e.g., 2) to work well on an average.

τ_a. The association strength threshold τ_a is required to define the association strength impact $\alpha(\phi)$ (Section 4.2.2). Fig. 18(c) illustrates the effect of different values of τ_a on the benchmark query `IQ5` on the IMDb dataset. The figure shows that, with very few example tuples, high τ_a is preferable, since it helps drop coincidental filters with weak associations. Similar to other parameters, with increased number of example tuples, the effect of τ_a diminishes.

Parameter	Default value	Description
ρ	0.1	Base filter prior parameter.
γ	2	Domain coverage penalty parameter.
τ_a	5	Association strength threshold.
τ_s	2.0	Skewness threshold.

Fig. 17. List of SQuID parameters with description.

τ_s. The skewness threshold τ_s is required to classify an association strength distribution as skewed or not (Section 4). Fig. 18(d) illustrates the effect of different values of τ_s on the benchmark query `IQ1` on the IMDb dataset. $\tau_s = N/A$ refers to the experiment where outlier impact was not taken into account (i.e., $\lambda(\phi) = 1$ for all filters). In this query, there were a number of unintended derived filters involving `certificate`, and a high τ_s helped to get rid of those. We also found a high τ_s to be very useful when we could not use a high τ_a due to the nature of the query intent (e.g., `IQ3`). However, too high value for τ_s is also not desirable, since it will underestimate some moderately skewed distributions and drop intended filters. Empirically, we found that moderate τ_s (e.g., 2–4) to work well on an average.

8. Comparative user studies

In this section, we present findings from our comparative user studies over SQuID and the traditional SQL-based mechanism. While prior work have conducted user studies for other PBE systems [22], query by example is a special category of programming by example that brings forth unique aspects and challenges. First, the traditional mechanism for retrieving relational data requires not only strong technical skills over the SQL language, but also familiarity with the structural organization of the data, called a schema. Schemas can be very complex, may contain domain-specific abstractions, differ from one database to the next, and could also get modified over time. As a result, even expert users with prior SQL experience can struggle to familiarize themselves with the schema of a previously unseen dataset, leading to difficulties in data exploration. Therefore, QBE needs to be studied from the perspective of users with varied levels of expertise, and the study needs to investigate the pain points specific to relational data access and exploration.

Second, the operational mechanisms in QBE systems fundamentally differ from those in general PBE systems. Traditional PBE approaches often rely on demonstration, where the mechanism to solve the intended task is demonstrated by the user. In contrast, in QBE, the user gives examples of the intended output and not the querying mechanism. Other PBE approaches rely on complete input–output specifications: the user needs to provide, typically small, sample inputs and outputs and the system infers their intended program. This mechanism is also not possible in a data exploration setting, where the input data is predetermined and typically large, and the user can only provide a small set of examples of their intended query output. Since the set of examples in the QBE setting is naturally incomplete, there is typically a much larger number of queries (programs) that could be compatible with them, compared to the general PBE setting; thus, the effectiveness of QBE systems needs to be explored with a targeted study.

Third, the setting of data exploration has two characteristics that can have significant impact in the performance of a QBE system: (1) Since the user needs to provide example records from the dataset at hand, domain expertise can have a bigger impact in the user experience than in the general PBE setting. (2) Data exploration tasks can be vague and subjective, and a strict specification is often hard, or even impossible, to derive, even by experts. This is a perspective not relevant to general PBE and has not been explored by prior studies.

We conducted two comparative user studies: (1) a controlled experiment study involving 35 participants, and (2) an interview study involving 7 interviewees to gain a richer understanding of users' issues and preferences. All participants and interviewees had varying levels of SQL expertise and experience, but were required to have at least basic SQL skills. Our studies focused on the task of data exploration and explored how SQuID compares against the traditional SQL querying mechanism, over a variety of objective and subjective data exploration tasks. Specifically, our study aimed to identify the most critical issues users face when interacting with the traditional SQL querying mechanism, to what extent a QBE system like SQuID can alleviate these challenges, how effective SQuID is over a variety of data exploration tasks, and what the possible pain points of SQuID are.

8.1. Dataset and baseline

In our comparative user studies, we studied how users perceive SQuID, compared to the traditional SQL querying mechanism, over a variety of subjective and objective data exploration tasks. We now provide an overview of the dataset we used in our studies and a brief description of SQL, the baseline which we compare SQuID against.

8.1.1. Dataset

For our comparative user studies, our goal was to emulate data exploration tasks in a controlled experiment setting. Generally, people explore data they are interested in and within a domain they are somewhat familiar with. Moreover, data exploration with QBE expects some basic domain familiarity, as users need to be able to provide examples. Therefore, our goal in selecting a dataset was to identify a domain of general interest, where most study participants can be expected to have a basic level of domain familiarity. Furthermore, the dataset needs to be sufficiently large to emulate the practical challenges that users face during data exploration. We selected the Internet Movie Database (IMDb),[7] which satisfies these goals. The IMDb website is well-known source of movie and entertainment facts, has over 83 million registered users and about 927 million yearly page visits.[8] The database contains information regarding over 10 million personalities, along with their demographic information; and about 6 million movies and TV series, along with their genre, language, country, certificate, production company, cast and crew, etc.

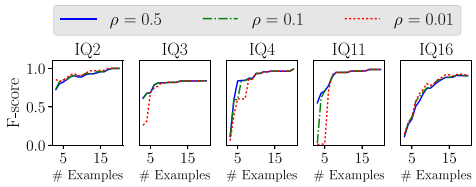
8.1.2. Structured query language (SQL)

The traditional way to query a relational database is to write a query in structured query language (SQL). SQL is one of the most widely used programming languages for handling structured data (54.7% developers use SQL [31]). It is specifically designed to query relational databases and has been used for over 50 years. SQL is a declarative query language and is primarily based on relational algebra. The SQL language consists of several elements such as clauses, expressions, predicates, statements, integrity constraints, etc. SQL has been implemented by different developers – such as Oracle, Microsoft SQL, MySQL, PostgreSQL, etc. – slightly differently; however, fundamentally, they all work the same way. For our comparative user studies, we picked PostgreSQL, which is a free and open-source relational database management system.

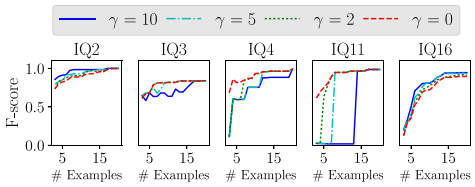
Relational databases usually organize data in a *normalized* form, to avoid redundancy. This is in contrast with the flat data format where all attributes of an entity are stored together within the same row. For example, the detailed schema of the IMDb database, split in 15 relational tables, is shown in Fig. 19. Here, the relation `movie` contains only three attributes about movies: a numerical record `id` (called *primary key*), a text attribute specifying the `title` of the movie, and the `production year` of the movie. However, information about associated genres of a movie is not present in the `movie` table. To figure out the genres of a movie, one would need to write a SQL

[7] IMDb: www.imdb.com/

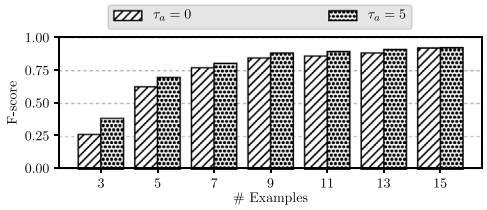
[8] IMDb.com Analytics: www.similarweb.com/website/imdb.com/



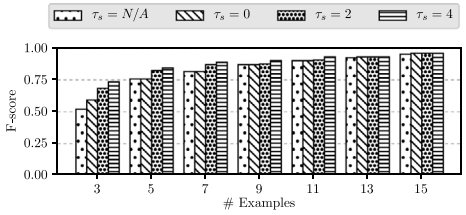
(a) Effect of different values for ρ for a few benchmark queries of the IMDb dataset.



(b) Effect of different values for γ for a few benchmark queries of the IMDb dataset.



(c) Effect of different values for τ_a for `IQ5` (IMDb).



(d) Effect of different values for τ_s for `IQ1` (IMDb).

Fig. 18. Effect of system parameters on SQuID's performance.

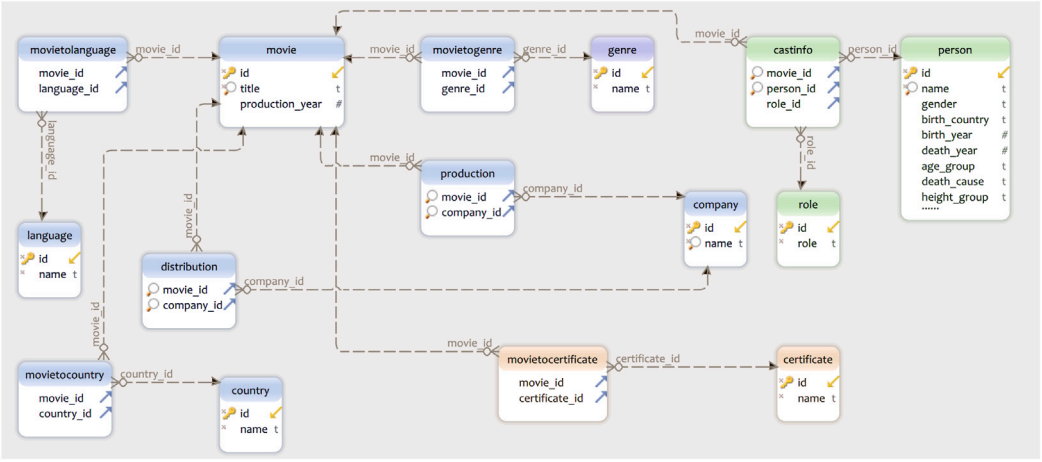


Fig. 19. Complete schema of the IMDb database with 8 main relations: `movie`, `person`, `genre`, `language`, `country`, `company`, `role`, and `certificate`; and 7 connecting relations that associate the main relations: `castinfo`, `movietogenre`, `movietolanguage`, `distribution`, `movietocountry`, `movietocertificate`, and `production`.

query to `JOIN` the tables `movie`, `movietogenre`, and `genre`. The query would also need to specify the *logic* behind this join, i.e., which rows in the `genre` table are relevant to a particular movie in the `movie` table. SQL is a relatively simple language with a limited set of operators (e.g., `SELECT`, `PROJECT`, `JOIN`, etc.). While this simplicity enables the users to learn quickly how to express easy intents using SQL (e.g., the SQL query `SELECT title FROM movie` would retrieve all movie titles), it comes at the cost that complex intents are hard to express in SQL. Specifically, the restrictions in the data organization (normalized schema) and the simplicity of the SQL operators make complex tasks harder to translate in SQL: it requires the users to specify the entire data retrieval logic. Overall, writing a successful SQL query for a data exploration tasks requires several skills: (1) familiarity with the database schema, (2) understanding of the table semantics, (3) understanding of the SQL operators, (4) knowledge of the SQL syntax, and (5) expertise in translating task intents to SQL.

8.2. Study design

In our user studies, our goal was to quantitatively compare the efficacy and efficiency of SQuID and SQL over a variety of data exploration tasks, while also gathering qualitative feedback from users regarding their experiences with the systems. To this end, we opted for two separate comparative user studies: (1) a controlled experiment study, with a fixed set of tasks, over a group of participants of sufficient size to support quantitative evaluation; (2) an interview study, with a flexible set of tasks, over a small group to gather qualitative user feedback. Due to the situation caused by the current COVID-19 pandemic, both studies were conducted online: the controlled experiment was conducted through a website, hosted on our university servers, and the interview study was conducted over Zoom.

For both studies, we provided the database schema (Fig. 19) and a graphical user interface with a text box, where the participants could write SQL queries to interact with a PostgreSQL database. For SQuID, we provided a graphical user interface to allow the participants to interact with the system (Fig. 20). We now proceed to describe the settings, design choices, and methods of our comparative user studies. We first describe our controlled experiment study over a user group of 35 participants, followed by our interview study with a smaller group of 7 interviewees.

8.2.1. Study 1: Controlled experiment study

Participants. For our controlled experiment study, we recruited students who were enrolled in an undergraduate computer science course

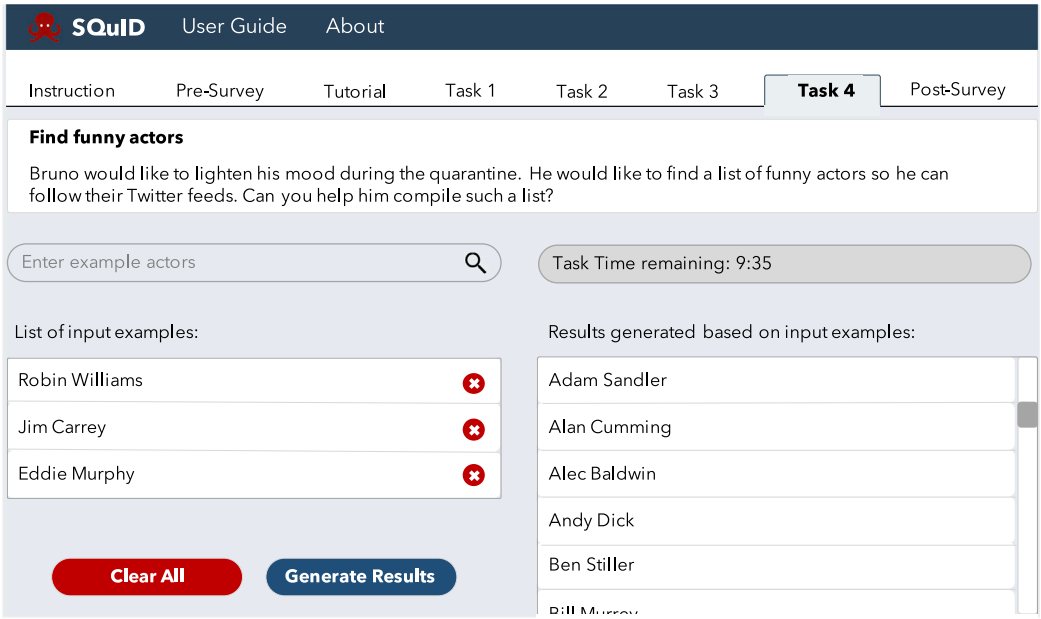


Fig. 20. The graphical user interface of SQuID used in our user study. The task description is at the top. The left panel allows the users to provide examples with an auto-completion feature. SQuID infers the user's intended query from the examples, executes it, and shows the results in the right panel. The result set contains more actors, but we only show the first five (alphabetically) here.

on Data Management Systems at our university during the Spring 2020 semester. The course offers an introduction to data management systems and the SQL language. This ensured that our study participants would have basic familiarity with SQL, which is required to compare the two systems: SQuID and SQL. We invited all 89 students enrolled in this course to take part in the study and 35 of them agreed to participate. We offered extra credit for study participation; students who opted to not participate were given alternative opportunities for extra credit. We labeled these participants P1–P35. The average grade the participants achieved in the course was 86.3 (out of 100), with a minimum grade of 45, and a maximum grade of 100; the standard deviation of the grades was 9.87. This indicates a broad range in our participants' SQL skills, which was one of our goals. While all of them had prior experience and exposure, some had only very basic skills (and failed the class) and some achieved advanced skills.

Tasks. We designed 4 data exploration tasks over the IMDb database. Our goal was to observe what challenges a set of diverse tasks poses to the participants and how the challenges vary based on the subjectivity of the tasks and the mechanism (SQuID or SQL) used to solve the tasks. To this end, we designed two objective tasks: (1) to find *Disney* movies and (2) to find *Marvel* movies; and two subjective tasks: (1) to find *funny* actors and (2) to find *strong* and muscular actors. We provided a detailed description for each task to the study participants. (Details are in the Appendix.)

Task assignment mechanism. Each participant was assigned all of the four tasks in the sequence: Disney, Marvel, funny, and strong. This order was enforced to ensure that they perform objective tasks first, which are easier, and then move to more complex and subjective tasks. We randomized task-system pairings to make sure that for each task, about half of the participants use SQuID while the other half use SQL. The task assignment mechanism was as follows: for each user, we randomized which system (SQuID or SQL) they are allowed to use for each task. Everyone did the tasks – Disney, Marvel, funny, strong – in that order, but there were two possible system assignment orders: (a) SQuID, SQL, SQuID, SQL, or (b) SQL, SQuID, SQL, SQuID. Each participant was randomly given one of these assignments. This resulted in randomized task-system pairings, with the constraint that each participant must solve one objective and one subjective task using SQL and the remaining two tasks (also one objective and one subjective) using SQuID. This mechanism also eliminated any potential order bias with respect to the treatment system as half of the participants interacted with SQuID before SQL, while the other half interacted with SQL before SQuID. Within each task (e.g., Disney), each participant used either SQuID or SQL to solve each task, but not both.

Study procedure. This study was conducted online and the participants took the study over the Internet on a specific website, hosted on our university servers. We sent out the URL of the website during recruitment. At the beginning of the study, participants were asked a series of questions about their familiarity with SQL. The questions asked the participants to provide answers using a 5-point Likert-scale ranging from "Not familiar (1)" to "Very familiar (5)". Next, there was a question asking them at what frequency they watch movies, followed by a questions about overall movie and actor familiarity where participants could select multiple options. After this survey, participants were given an interactive tutorial, which was divided into two sections, walking them through the steps to obtain results with both SQuID and SQL. The tutorial took about 2–5 min to complete. After the tutorial, the participants started the tasks. They had 10 min for each task, but could finish before the time was up if they chose. Participants were asked to avoid using Internet search, but if they did, they were encouraged to report it. After each task, the participants were asked to answer a post-task survey with two questions: the first one was about the difficulty of the task where the participants had to provide answers using a 5-point Likert-scale ranging from "Very difficult (1)" to "Very easy (5)"; and the second one was about their satisfaction with the results where the participants had to provide answers using a 5-point Likert-scale ranging from "Very unsatisfied (1)" to "Very satisfied (5)". After completing all four tasks, the participants were asked to answer four survey questions: the first one was regarding their preferences between SQuID and SQL where the participants had to provide answers using a 5-point Likert-scale ranging from "Definitely SQL (1)" to "Definitely SQuID (5)"; the second one was about usability comparison between SQL and SQuID where the participants had to provide answers using a 5-point Likert-scale ranging from "SQL was a lot easier (1)" to "SQuID was a lot easier

Interviewee ID	Gender	Country of origin	Program level	SQL expertise	Area of specialization
I1	Female	Greece	2nd year PhD	Medium	Data management
I2	Male	India	3rd year PhD	Low	Natural language processing
I3	Male	Hong Kong	2nd year MS	High	Systems
I4	Female	China	5th year PhD	High	Data privacy
I5	Male	India	4th year PhD	High	Theory and data management
I6	Female	Japan	2nd year PhD	Medium	Data privacy
I7	Male	USA	4th year PhD	High	Data privacy

Fig. 21. Demographic and experience details of the interviewees who participated in our interview study.

(5)"; the third one was about satisfaction with results obtained using SQuID where the participants had to provide answers using a 5-point Likert-scale ranging from "Very unsatisfied (1)" to "Very satisfied (5)"; and the fourth one was about accuracy of the results obtained using SQL where the participants had to provide answers using a 5-point Likert-scale ranging from "Very inaccurate (1)" to "Very accurate (5)".

Data collection. During the study, we collected all survey responses and all inputs the participants provided to the systems. Specifically, for SQL, we collected all their queries, including any intermediate queries that they used to reach their final query; for SQuID, we collected all the examples they provided, along with the revision history (addition or removal of examples). We stored all this information in JSON format.

Data analysis. During our data analysis, we extracted the JSON data programmatically through Python scripts and implemented custom functions to programmatically analyze the data. To quantitatively evaluate the tasks performed by the participants, we compared their results against the ground-truth results. We collected the ground-truth data from publicly available lists on the IMDb website. For the objective tasks (Marvel and Disney), we determined the ground truth by selecting one list for each. For the subjective tasks (funny and strong), we compiled a list by combining seven different lists for each. We selected lists that meet the following criteria: (1) they have a number of entries that is representative of the task (e.g., there are more than five Marvel movies, thus the list should contain more than five entries), (2) they are frequently viewed, and (3) they contain entries that match the task objectives. For instance, we collected a list of 300 funny actors, which was compiled from 7 shorter lists of funny actors. One of these lists, titled "Funny Actors", has over 400,000 views, and includes 60 well-known comedians including Jim Carrey, Robin Williams, Eddie Murphy, Mel Brooks, and Will Ferrell.[9] We provide all the lists we used in the Appendix.

8.2.2. Study 2: Interview study

We conducted a comparative interview study to gain richer insights on users' behavior, their preferences, and any issues they faced while solving the data exploration tasks using both systems.

Interviewees. We recruited 7 interviewees for this study by targeting a diverse set of computer science graduate students directly working or collaborating with the data management research lab at our university. Out of the 7 interviewees, 4 were male and 3 were female; 6 of them were international students; and their ages ranged from 25 to 30 years old. All of them had experience using SQL for at least one year; however, their expertise varied from moderate to expert. We label the interviewees I1–I7. We provide further details on the interviewees in Fig. 21.

Tasks. For this study, we asked the interviewees to pick one objective task from the following list: (1) Disney movies, (2) Marvel movies, (3) animation movies, (4) sci-fi movies, (5) action movies, (6) movies by an actor of their own choice, or (7) movies by a country of their own choice. We also asked them to select one subjective task form this list: (1) funny actors, (2) physically strong actors, or (3) serious actors. The variety of tasks allowed interviewees to pick tasks based on their interests and enabled us to observe how the two systems compare over a variety of data exploration tasks. This study was within-subject, i.e., all of the interviewees were required to use both the mechanisms (SQuID and SQL) to solve each task.

Study procedure. For each interview, two of our research team members were present, one as primary to lead the interview and ask questions and another as secondary to take notes and ask potential follow-up questions. At the beginning of the study, we provided them the URL of the study website over the chat feature of Zoom. During the study, the interviewees first completed an interactive tutorial and then they were asked to pick two tasks. The interviewees were then asked to solve each task using both SQuID and SQL, so that they can directly contrast the two systems. We asked them to complete each task first using SQuID and then using SQL, so that the examples they would provide while using SQuID would be free from biases due to observing the results from their SQL query outputs. We did not expose the query that SQuID generates through the SQuID interface, and, thus, avoided biases when the interviewees were completing the SQL tasks. The interviewees followed a think-aloud protocol and shared their screen over Zoom during the study. They were observed by two interviewers who also asked open-ended questions to the interviewees on completion of each of the two tasks using both systems. The questions were intended to gather information on which of the two systems the interviewees prefer, under what circumstances they prefer one over the other, and the justification of why they do so. They were also asked what challenges they faced while using the systems and whether some particular task exacerbated these challenges. Finally, they were asked what type of results they prefer during data exploration: specific or generic.

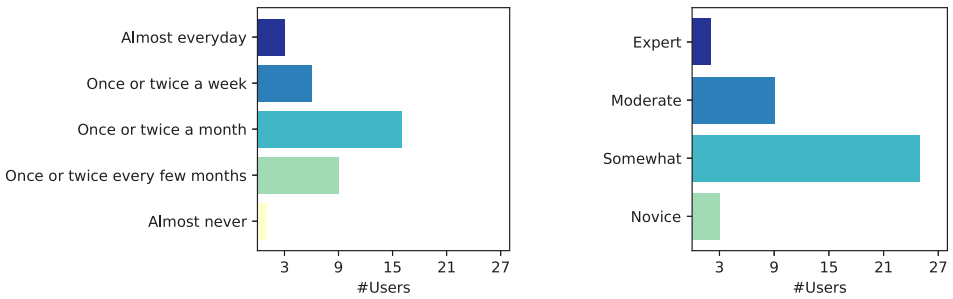
Data collection. We recorded all interview sessions. The 7 interviews summed to 467 min. On average, each interview lasted about 67 min, with the shortest interview lasting 43 min and the longest one lasting 77 min. Upon completion, we replayed the interview recordings, manually transcribed the responses, and stored them as plain text in a spreadsheet, resulting in 119 responses in total.

Data analysis. We thematically analyzed the responses using our coding software (spreadsheet). Two independent coders from our team independently coded the data. The following six themes emerged after several rounds of analysis: (1) struggle in task understanding, (2) struggle in familiarizing oneself with the schema while using SQL, (3) difficulties with writing syntactically correct SQL queries, (4) struggle with solving vague/subjective tasks using SQL, (5) struggle due to lack of domain familiarity while using SQuID, and (6) preference between precision and recall of the results. Inter-coder reliability was 0.98, calculated using Krippendorff's alpha.

8.3. Quantitative results from controlled experiment

We now present the quantitative results of the controlled experiment study, summarizing our findings.

[9] Funny Actors: https://www.imdb.com/list/ls000025701



(a) Movie-watching frequency. (b) Overall knowledge of movies and actors.

Fig. 22. Domain knowledge of the participants.

Participants had basic domain knowledge and SQL familiarity. The distribution of self-reported movie-watching frequency among the participants is shown in Fig. 22(a), with the most common response being 'once or twice a month', followed by 'once or twice every few months'. The responses regarding actor and movie familiarity are summarized in Fig. 22(b): a vast majority of the participants (25 out of 35) reported that they were 'somewhat' familiar with movies and actors. This validates our choice of the IMDb database for conducting the study, as indeed, we observed sufficient domain knowledge among the participants. Regarding SQL expertise, all 35 participants reported being very familiar with easy SQL queries and 34 reported being very familiar with moderately complex SQL queries. When asked regarding familiarity with complex SQL queries, 27 participants reported being very familiar, 6 were unsure, and 2 were unfamiliar.

SQUID *is generally more effective than* SQL *in generating accurate results.* To quantitatively measure the quality of the results produced by both SQUID and SQL, we checked them against the ground-truth results (discussed in Section 8.2.1). We used three widely used correctness metrics to quantify the result quality: precision, recall, and F1 score. These metrics capture different aspects: precision captures "preciseness", i.e., the fraction of retrieved tuples that are relevant; recall captures "coverage", i.e., the fraction of relevant tuples that are correctly retrieved; and F1 score – which is a harmonic mean of precision and recall – maintains a balance between them.

On average, we found SQUID to be more effective in generating accurate results than SQL (Fig. 23). For all four tasks, on average across participants, results obtained with SQUID achieved significantly higher precision than the results obtained with SQL. SQUID achieved higher recall than SQL for the two objective tasks (Disney and Marvel). While SQUID's recall for the subjective tasks (Funny and Strong) was lower than SQL, note that SQL's precision for those tasks was close to 0. This is simply because the SQL queries the participants wrote for those tasks were very imprecise and returned a very large number of results (e.g., all actors in the database). While such general queries can happen to contain a large portion of the correct results (hence the high recall), they contain an extremely large number of irrelevant results making them poorly suited for this retrieval task. In terms of F1 score, SQUID always achieved higher values than SQL implying its effectiveness over SQL for generating more accurate results. The result of t-tests for these findings are shown in Fig. 24. Out of the 12 findings, 7 are statistically significant with a p-value less than 0.05.

Participants were more efficient with SQUID *than* SQL. SQUID helped the participants solve the tasks more quickly (Fig. 25(a)) and with fewer attempts (Fig. 25(b)) than SQL. On average, the participants were able to solve the tasks using SQUID about 200 s faster than when using SQL. Participants were also able to solve the tasks with about 4 fewer attempts while using SQUID compared to SQL. The results of t-test of these findings, shown in Fig. 26, signify that most are statistically significant with a p-value less than 0.05.

Participants generally found SQUID *easier to use and more satisfying, but still preferred* SQL. Figs. 27(a) and 27(b) show self-reported overall satisfaction with the results produced by SQUID and SQL, respectively. Generally, participants found the results produced by SQUID more satisfying than the results produced by SQL. Out of the 35 participants, 23 were somewhat or very satisfied with SQUID. In contrast, 18 reported that the results produced by SQL were somewhat or very accurate. However, we found that the self-reported satisfaction does not correlate with the actual correctness of the results (measured in terms of precision, recall, and F1 score), and in fact, the participants generally did better with SQUID than SQL, although they did not always realize it. Fig. 27(c) shows self-reported overall evaluation comparing SQUID and SQL in terms of ease of use. Out of the 35 participants, 19 reported that SQUID was easier, 6 reported that they had the same level of difficulty, and 10 reported that SQL was easier.

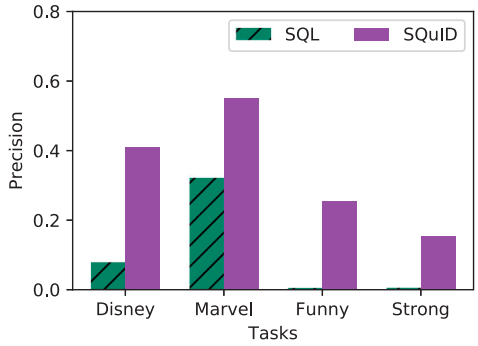
However, despite reporting that SQUID was easier to use and the results were more satisfying, the participants were still leaning towards SQL as a preferred mechanism for data exploration. Fig. 27(d) shows self-reported overall preference between SQUID and SQL, where 11 reported that they would prefer SQUID while 19 reported that they would prefer SQL. Five participants reported no preference.

8.4. Qualitative feedback from interview study

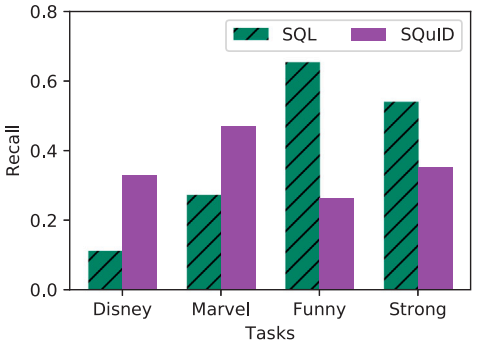
We now report the results of our interview study and describe six main themes that emerged from our analysis.

Studying the schema is challenging, even for SQL *experts.* All seven of our interviewees from the interview study commented that it was difficult to become acquainted with the database schema. "As a user, I have to *explore* the schema", I1 said. I1 continued, "The query itself was not complicated. It was time consuming to get familiar with the schema itself. Even for experienced users, reading through the schema and getting acquainted to [it] ... takes time". When asked about the comparison in difficulty between writing the SQL query and understanding the schema, I3 said "Looking at the schema diagram was harder. I kept going back and forth trying to understand it". Understanding the schema may be complicated not only because it can be difficult to learn what keys connect the tables, but also because it may be hard to interpret the structure of the individual tables. I5 said, "I think it was pretty hard because I was not sure where to look for comedy based on actors. I was thinking that [the] Role [table] might have the attribute, but it didn't. Then I had to go through joining five tables!"

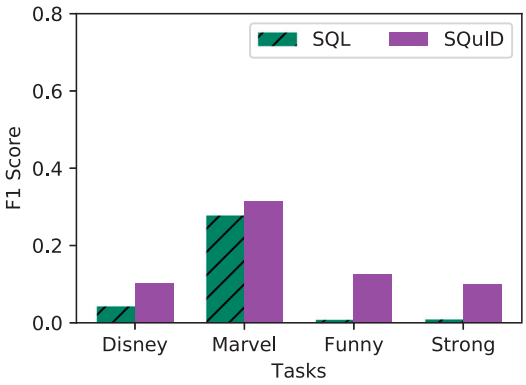
SQL *requires stricter syntax, which makes writing queries difficult.* All interviewees struggled to a varying degree to write a SQL query because of different issues; e.g., some of them could not figure out the correct spelling of attributes. For instance, one queried for the genres 'scifi' and 'comedic', neither of which exist in the database. I4 said, "The difficult part was to get the accurate predicate for the query, and I had to [explore the database] for that". SQL requires strict string matching, which can be extremely difficult to overcome for someone



(a) SQUID achieves higher precision than SQL for all tasks.



(b) SQUID achieves higher recall than SQL in two tasks.

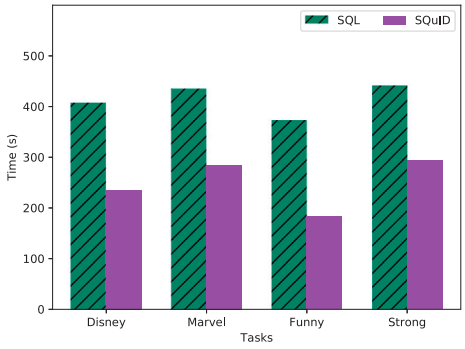


(c) SQUID achieves higher F1 score than SQL for all tasks.

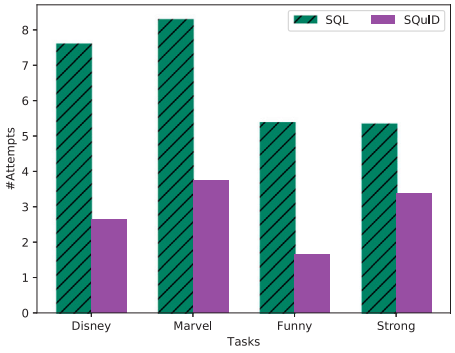
Fig. 23. Comparison of SQUID vs. SQL in terms of average precision, recall, and F1 score.

	Precision		Recall		F1 Score	
Task	p-value	t	p-value	t	p-value	t
Disney	**0.004**	3.0781	**0.0389**	2.1457	0.151	1.468
Marvel	0.1047	1.6669	0.0588	1.9554	0.7195	0.3621
Funny	**0.0001**	4.3845	**0.0042**	-3.0751	**0.0**	8.6225
Strong	**0.011**	2.6935	0.1751	-1.3859	**0.0**	6.4942

Fig. 24. t test results for precision, recall, and F1 score. Out of 12 findings, 7 are statistically significant. In all cases, df = 33.



(a) Time required to solve each task.



(b) Number of attempts required to solve each task.

Fig. 25. Comparison of SQL vs. SQUID in terms of effort (average time required and average number of attempts) for solving the same set of tasks.

who is unfamiliar with the database constants and SQL syntax. While it is possible to query a table and view its content to see how the names are spelled, very few interviewees did this. It appears that the ability to write a SQL query is based on experience and recent exposure to SQL. Interviewees noted that they do not use SQL on a daily basis – some even said they had not used SQL in months – thus, it was difficult to recall specific syntactic rules. For instance, two of the interviewees – who had relatively lower SQL expertise – could not remember the requirements for joining tables. I7 had to use Google to help with this syntax, and I2 did not recall that SQL could join more than two

	Task completion time		#Attempts	
Task	p-value	t	p-value	t
Disney	**0.0014**	-3.5	**0.0**	-4.7578
Marvel	**0.0146**	-2.5767	**0.0008**	-3.6985
Funny	**0.0008**	-3.7105	**0.0007**	-3.7441
Strong	**0.0132**	-2.6206	0.0595	-1.9518

Fig. 26. t test results for task completion time and number of attempts. Out of the 8 findings, 7 are statistically significant. In all cases, df = 33.

tables. I5 said, "I was making a lot of mistakes about where to have the underscores, where to not have underscores, and for those things I had to look through the [schema] multiple times". An interesting note, I6 spent the vast majority (over 9 min) attempting to find the name 'Japan' in the database, and spent less than 1 min writing the actual query. SQuID reduces the need to recall exact spelling by providing an auto-completion feature as the user types examples. Although it does not provide an auto-correct if the name is misspelled, the auto-completion feature allows the users to type what they know and scroll through the suggestions until they find the intended name. We observed several of our interviewees initially spelled a movie name incorrectly, but they were helped by the auto-completion feature. For example, I2 initially typed 'Spiderman' in the search bar, but the title is spelled 'Spider-Man' in the database. I2 was able to correct the spelling when he typed 'Spider' in the search box and autocomplete showed the entire title. The search bar also helped I5 who noted, "If I was missing some spellings, there were some suggestions".

SQL *requires parameter tuning for subjective tasks;* SQUID *alleviates this.* Some exploration tasks can be subjective and inherently vague, e.g., defining a "funny" actor *precisely*. How many comedies, exactly, does an actor have to star in before they are considered funny? These questions have no clear answers, and such parameters can vary from person to person and from day to day. In practice, it may be very difficult, if not impossible, to think of objective measures for a subjective concept, which makes subjective tasks very complicated to specify with SQL. I2 said, "Even if I forget about syntax ... figuring out how to go about writing the pseudocode query for funny actors [is difficult]". One of the most common blunders of interviewees who used SQL to find "funny" actors was to query all actors who had been in some comedy movie. I3 was the first to acknowledge this. "I had to play around with a lot of smaller queries", he said, "to get the one that I eventually had, which I was still not satisfied with. It seems like I pulled many actors and actresses that happened to be in some comedy". I3 elaborated, "Vague tasks are generally a lot more open to interpretation. Coding up a query that meets someone's vague specifications [is] hard ... It was very hard to nail down what the correct definition of funny is". I4 also recognized that vague tasks are difficult to define. She even said, "This probably isn't a query that I should write in SQL!" She continued, "strong and muscular are very vague descriptors, and SQL needs clear rules. I have to use genre as a proxy, and that makes the query very nasty".

On the other hand, SQuID can interpret complex parameters without any involvement from the user, sparing them the mental burden of defining and implementing a complex query. I4 also said, "In order to write a SQL query, you need to understand the schema well, know your data well, and know your question well ... But if the task is exploratory and you only have a vague idea in mind, like 'strong actors' ... it would be very hard, if not impossible, to write a SQL query". Indicating how SQuID helped in the subjective tasks, I3 said "SQuID is a lot more user-oriented. You could just put in some actor names and it would infer what you really want".

SQUID *produces precise results, which is preferred for data exploration.* We asked interviewees whether they would prefer a long list that includes all relevant names, but may also include many irrelevant names (high-recall) or a shorter list that includes exclusively relevant names with very few irrelevant names, but may miss some relevant names (high-precision). Six out of seven interviewees reported that they would prefer having a shorter list with higher precision, while one interviewee had no preference. "I think I'm okay with not having all Marvel movies listed here", I2 said, "but I definitely don't want anything outside of Marvel movies. It's fine that [the results] are missing some Thor movies. I wouldn't have liked it if there were movies from DC [Comics] in here". Comparing the SQL results to the SQuID results, I5 said, "I think the [SQuID] results were not too few but not too many. It was easily understandable, and I could actually see if these were actors I was looking for ... The [SQL] results were just too many, and most of the names I didn't know, so it was not easy to find the names that I was looking for". I6 said, "I prefer a shorter list because if there are too many movies listed, then probably, it would be overwhelming and I could not say if the results are right".

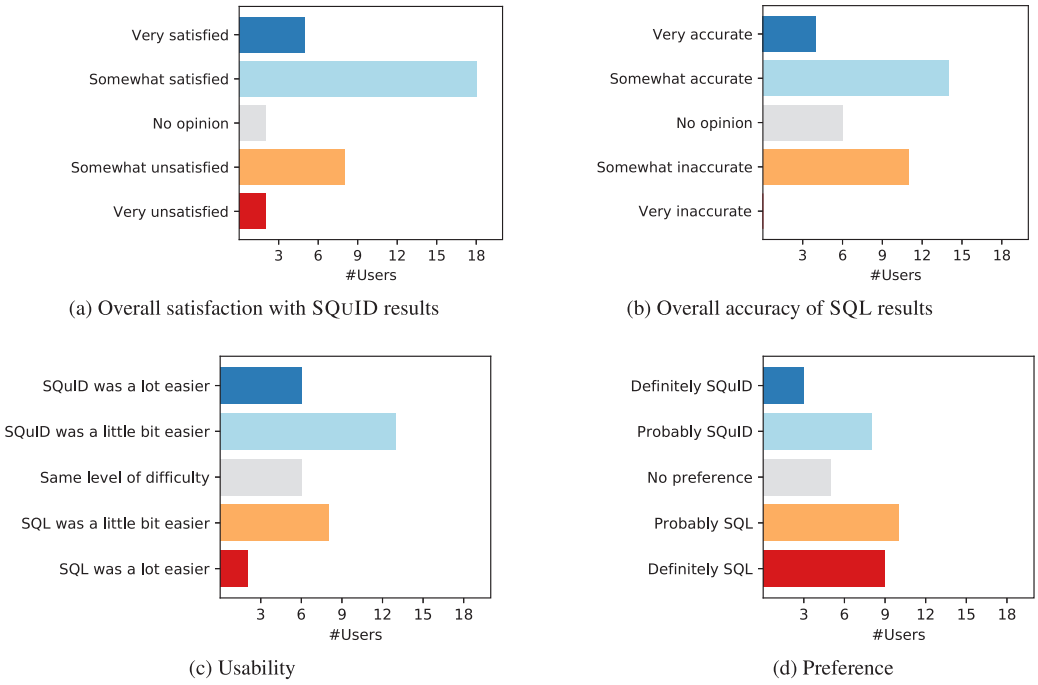
SQUID*'s interactivity helps users to enrich examples.* Three interviewees mentioned that the results produced by SQuID helped them think of more examples in an iterative process. I6, who struggled to think of examples, was able to think of only three sci-fi movies, but when she saw 'Avatar' in the list of results, many other ideas came to her mind. Even if the intermediate results (the first or second round of results generated) were not all intended, some of them were useful in reminding the interviewees of relevant examples. For instance, I2 said, "SQuID was [nice] because it was slightly interactive. I could look at the results and update my examples". During a task, I7 said, "[The results are] useful because now I can use Guardians of the Galaxy". I7 later added, "I think when I gave the first few examples, it gave me some results and that helped me think of more that I was looking for, and it eventually did complete the task". SQuID's results reminded the interviewees of examples that had not been on their mind, but were nonetheless relevant. I3 said, "I saw the movie Transformers, and that's something I had in my mind, but it did not occur to me when I was entering the examples. There were a bunch of other movie names [like that]". Since SQuID can provide serendipitous, but helpful, intermediate results, the user's lack of domain familiarity can still be alleviated to some extent.

Domain familiarity is crucial to evaluate the results, for both SQUID *and* SQL. SQuID requires a basic familiarity with the domain. For those who struggle to think of even one relevant example, like I6, SQuID presents a unique challenge. All interviewees could easily think of a few examples that fit the task, but they struggled beyond that. I7 said, "It was very easy to come up with two or three, but the more examples I had to give the harder it became". Two interviewees suggested that SQuID adopt an interactive system where it would ask the user whether or not a particular result was relevant on a case-by-case basis. This could alleviate some of the difficulty of thinking of relevant examples.

Furthermore, users who possess very little knowledge of the domain may be unable to recognize the results, and thus would be incapable of verifying them. But this is true for both SQuID and SQL. It was not uncommon for the interviewees to tell us that they could hardly recognize the names in the results, especially for SQL. I1, for instance, said, "Honestly, I don't recognize any of the results". This, apparently, was partly due to the large number of results returned by SQL, where there is a high chance that there will be unfamiliar names. Most people are only familiar with a relatively small subset of actors, rather than the entirety of the IMDb database. This made it difficult for the users to evaluate the results produced by both SQuID and SQL.

9. Discussion, limitation, and future work

In this section, we discuss key takeaways of our user studies, the limitations of SQuID, and future directions.



(a) Overall satisfaction with SQuID results

(b) Overall accuracy of SQL results

(c) Usability

(d) Preference

Fig. 27. Comparison of SQuID vs. SQL in terms of various metrics (self-reported).

9.1. Key takeaways

We summarize significant findings from the quantitative and qualitative analysis of our comparative user studies and highlight the key takeaways below.

SQuID *alleviates* SQL *pain points: schema complexity, semantic translation, and syntax.* From our interviews, we identified three key pain points of the traditional SQL querying mechanism, all of which are removed when using SQuID:

Schema complexity. One significant difficulty that we observed during the use of SQL was the requirement of schema understanding. To issue a SQL query over a relational database, the user must first familiarize themselves with the database schema [7,32]. The schema is often complex, like the IMDb schema shown in Fig. 19, and understanding it requires significant effort. The user also needs to correctly specify the constant values (e.g., `Comedy` and not `Comedic`), name of the relations (e.g., `movietogenre` and not `movie_to_genre`), and name of the attributes (e.g., `id` and not `movie_id`) in the SQL query. Moreover, some attributes reside in the main relation (e.g., `person.name`) while others reside in a different relation (e.g., names of a movie's genres reside in the relation `genre` and not in the relation `movie`). From a closer look at some of the user-issued SQL queries, we observed futile efforts to guess keywords, incorrectly trying values such as "comedic", "superhero comics", and "funny", which do not exist in the database and result in syntax or semantic errors. In structured databases, if one does not know the exact keywords, they end up issuing an incorrect SQL query, which returns an empty result. In contrast, SQuID frees the user from this overhead as it leverages the database content and schema and associates it automatically with the user-provided examples.

Semantic translation. After studying the schema, the next task was to translate the task's semantics formally to a language (e.g., SQL) that computational systems understand. While this is relatively easy for objective tasks (e.g., finding all movies produced by Disney), the same is not true for subjective tasks (e.g., finding all "funny" actors). As our qualitative feedback indicates, expressing subjective or vague tasks is hard in any formal language, not only in SQL. For example, for the task of finding all "funny" actors, even the SQL experts struggled to encode the concept "funny" in SQL. Many participants wrote a SQL query to retrieve all actors who appeared in at least one movie whose genre is `Comedy`. However, upon observing the output of such an ill-formed query, they were not satisfied with the results. This is because appearing in only one comedy movie does not necessarily make an actor funny. Usually, actors who appear in "many" comedy movies are considered funny. The key struggle here is to figure out what is the right threshold for "many", i.e., in *how many* comedy movies should an actor appear to be considered "funny". In contrast, SQuID is able to discover these implicit constants from the user-provided examples. For retrieving funny actors, SQuID learns from the user-provided examples what is the usual number of comedy movies all the example actors appeared in, and subsequently, uses that number to define the notion of "many". For instance, for Example 1.3, SQuID inferred that appearing in 40 comedy movies is sufficient for an actor to be considered funny. This parameter (40) was automatically inferred based on the user-provided examples: SQuID automatically discovered that each example actor appeared in 40 or more comedy movies in the IMDb database.

Language syntax. SQL is a programming language with several operators and keywords, and similar to all programming languages, SQL also requires strict syntax. While issuing a SQL query, even a minor syntactic error will result in complete failure and will return no result. Moreover, the syntax error messages that the SQL engine provides are often ambiguous and confusing to novice users. We observed that one of our interviewees could not recall the correct syntax of the `JOIN` operation. This stringent requirement of syntax poses significant hurdles to novice and even intermediate SQL users. In contrast, SQuID completely bypasses SQL, eliminating this challenge.

SQuID *is generally more effective than* SQL *and boosts efficiency.* In our controlled experiments, we noted that SQuID is generally more effective than SQL in deriving accurate results. For objective tasks, we found that SQuID outperforms SQL in all three correctness metrics—precision, recall, and F1 score. However, it is important to highlight that our interviewees noted that SQuID is particularly useful and preferable to SQL for *subjective* tasks. This does not contradict our quantitative analysis. While SQL has higher recall than SQuID for subjective tasks,

SQuID achieves much higher F1 scores, because SQL's precision for these tasks is close to 0. This is because an extremely general SQL query (e.g., one that returns all the data) may have very high recall, but it will not be useful for the exploration task that expects targeted results. Furthermore, SQuID significantly boosts the user's efficiency in data exploration. This was confirmed by our controlled experiment study where we found that participants achieved their goal much faster (in about 200 fewer seconds) and with less effort (with about 4 fewer attempts) while using SQuID compared to SQL.

SQuID*'s pain point and remedies: lack of domain expertise.* Lack of domain knowledge is a handicap for SQuID, as it requires at least a few initial examples for its inference. This is a general issue with all query-by-example mechanisms [7,33]. However, even when the user lacks domain knowledge, they can use alternative mechanisms – such as keyword search, Internet search, or very basic SQL queries (when the user has some SQL familiarity) – to come up with some initial examples. In contrast, when a user does not know SQL, learning it from scratch takes significant time and effort. While SQuID's by-example paradigm can help both expert and novice users alike, in general, programming-by-example systems are most beneficial when domain knowledge outweighs technical knowledge and experience [22]; otherwise, a hybrid system is more desirable. However, lack of domain knowledge is a problem for SQL as well. Without basic knowledge over the data domain (e.g., what are the entities and what are their properties), understanding the schema can be harder. Furthermore, without sufficient domain knowledge, debugging SQL queries, i.e., validating whether the user-issued SQL queries are correct or not, based on the results, is also challenging.

SQuID *promotes serendipitous discovery, aiding in data exploration.* SQuID is *interactive* in a sense that the users can revise their examples based on the results and even use some of the results as examples in the next iteration. A number of interviewees mentioned that by looking at the results that SQuID generated from their initial examples, they were able to come up with new examples. Moreover, when their examples contained some unintentional bias – e.g., while retrieving Disney movies, they only provided examples of recent movies – they were able to receive implicit feedback of that bias by SQuID as the results SQuID generated reflected the same bias. This feedback mechanism helped them revise their examples accordingly. In contrast, SQL does not offer such interactivity or feedback mechanism. While some interviewees used subqueries of the main query to view some intermediate results, this was just for the purpose of verifying the correctness of the main query. In contrast, SQuID's natural interaction and feedback mechanism offers additional help to the users. This makes SQuID particularly suitable for the task of data exploration. SQuID often promotes *serendipity* in the results—providing a good balance between *exploration* (serendipitous, surprising, and novel discovery) and *exploitation* (similar to the examples)—which is a desired property during data exploration.

SQuID *is particularly useful for solving complex and subjective tasks.* The specific properties of SQuID, specifically interactivity, providing feedback, and promoting serendipitous discovery, make it a significantly better choice for solving subjective tasks that are usually ambiguous and vague, and are very hard to solve using SQL. For example, in our studies, we used "strong actors" or "funny actors" as two examples of subjective tasks. Participants of both our controlled experiment study and interview study found thinking of examples easier than expressing their intent using SQL, especially for subjective tasks. Our results indicate that SQuID provides an easier mechanism for data retrieval and helps users overcome the difficulty of writing overly complex SQL queries for subjective tasks. In contrast, for objective tasks, we found both SQuID and SQL equally effective, given the user has basic SQL expertise.

Trust on a system depends on prior exposure, expertise, type of the tasks, and system explainability. During our controlled experiment, we wanted to measure how much the participants trust the mechanism that produces the results by asking the questions: "how well do you think SQuID did in generating the desired results?" and "how accurate were the SQL results?" While some participants reported that they were more satisfied with the results produced by SQuID than SQL, interestingly, many of them reported that they prefer SQL over SQuID even though they generally did better with SQuID (Fig. 27(d)). This result is in line with prior work that compared a PBE tool against traditional shell-scripting and found that despite performing better using the PBE tool, users tend to trust the traditional shell-scripting more [22]. We validated this by checking against ground-truth results where SQuID groups achieved results with higher precision (more specific) and F1 score (more accurate), as shown in Fig. 23.

Since the participants performed better when using SQuID compared to SQL, we interpret their preference for SQL to be due to three possible sources of bias: (1) *Familiarity:* The participants were at the time taking a course on relational databases and SQL, which may have artificially increased their confidence in their SQL skills. They had prior experience with SQL, but were experiencing SQuID for the first time through the study. (2) *Explainability:* SQL exposes the precise mechanism (the code) that produces the results, while we did not provide participants with an explanation of the inner workings of SQuID nor exposed the query it produces. (3) *Domain expertise:* Low domain expertise poses a hurdle in producing examples for SQuID; we posit that the users may consider SQL a more versatile mechanism for such circumstances.

We further investigated the issue of trust during our interview study by asking all our interviewees the question: "Which of these two systems, SQuID or SQL, do you trust more?" We expected SQL experts to trust SQL more, but did not observe any strong trend. Rather, the interviewees mentioned that for objective tasks, they were more confident about the SQL queries they wrote, and hence, they trusted SQL more. In contrast, for the subjective tasks, they reported that they trusted the results produced by SQuID more, as for the subjective tasks, the most common complaint was that SQL produced too many results (less specific) and perhaps retrieved the entire database content. Ultimately, SQuID can also provide explanations, by exposing the SQL query it synthesizes in order to generate the results and the underlying mechanism used to synthesize the query. We shed more light on this in the future work.

SQuID *is easy to learn.* A desired property for any system is *learnability*: how easy it is to get used to the system. From our study, we found that it was very easy for the participants to learn how to use SQuID almost instantly. SQuID's interface is intuitive and both novices and experts learned how to use it, just by observing its behavior. In contrast, when participants did not know how to write certain classes of SQL queries, they simply gave up and mentioned that they cannot express their logic in SQL. This is particularly significant considering that all our study participants and interviewees had prior exposure to and experience with SQL, while this was their first experience using SQuID.

9.2. Limitations and future work

Our study results indicate that SQuID effectively helped users with various levels of SQL familiarity perform their tasks faster and more efficiently. However, our work explored only one example of QBE systems and recognizably with a limited number of participants. Additional work is needed to study the impact of QBE systems further. While our goal was to draw a comparison between traditional SQL querying and QBE systems, additional studies might investigate how complete novices (users with no SQL expertise) use QBE systems. Furthermore, future studies can expand the list of tasks to better tease apart the impact of using QBE systems for various task types. From the

interviewees' feedback, we extracted a few directions for future work to improve user experience while using QBE systems:

Exposing internal mechanism for explainability. QBE systems like SQuID can explain how the SQL queries were synthesized from the user examples by exposing the particular semantic similarities that the system discovers across the examples, and its confidence in each similarity being intended. This can also guide users in revising their examples to emphasize borderline semantic similarities that SQuID missed, or diversify examples to avoid coincidental similarities among the examples.

Tuple suggestion to enrich examples. A few interviewees reported that it would be helpful if SQuID could suggest a few tuples that the user may consider adding to the examples. Such a tuple-suggestion mechanism will help the users supply additional examples and diversify the examples, in case the users lack domain knowledge.

Interaction with the results for feedback. Another direction of future work is to allow the users to interact with the results produced by QBE system: the user will accept or reject a few result tuples which will act as feedback to the system. This will help QBE system learn the user intent better.

Extensive user study. More extensive user studies are needed in the future to evaluate all these additional features and determine whether they contribute positively to the users' trust and satisfaction in QBE systems.

Beyond the ones extracted from the user study, there are several possible improvements and research directions that can stem from our work, including smarter semantic context inference using log data, example recommendation to increase sample diversity and improve abduction, techniques for adjusting the depth of association discovery, on-the-fly αDB construction, and efficient αDB maintenance for dynamic datasets.

10. Related work

SQuID was first demonstrated in a 2018 conference [34] and later described in details by the authors in a 2019 conference paper [35]. This article extends the conference paper by including (1) additional details regarding the probabilistic abduction model, specifically, modeling query prior (Section 4.2.2), (2) experimental results on the impact of the system parameters on SQuID's performance (Section 7.7), (3) results and analysis of two new comparative user studies (Section 8), (4) extensive discussion on key takeaways of the user studies indicating SQuID's limitations and scope of future extensions (Section 9), and (5) an extensive discussion on how SQuID contrasts with prior work (Section 10). The new comparative user study on real users is foundational to our understanding of how users interact with relational databases for data exploration tasks. Especially, from the comments of the interviewees during our interview study, we gained an inside look at how inexperienced users interact with QBE tools as well as their thought processes behind writing complex queries with traditional query interfaces.

We proceed to provide an overview of the related work. Then we contrast SQuID with prior art, where we provide an extensive discussion to contrast SQuID against existing semi-supervised machine learning approaches and data cubes.

10.1. Overview of related work

Programming by example (PBE) paradigm is based on the intuitive premise that users who may lack or have low technical skills, but have expertise in a particular domain, can more easily express their computational desire by providing examples than by writing programs under strict language specifications. This is in contrast with traditional program synthesis [36,37], which requires a high-level formal specification (e.g., first-order logic) of the desired program. Example-driven program synthesis has been effectively used for a variety of tasks, such as code synthesis for data scientists [38], data wrangling [39], integration [40], extraction [41,42], transformation [33,43], and filtering [44]; data structure transformation [45]; text processing [46], normalization [47], and summarization [48]; querying relational databases [7], and so on.

Many PBE approaches have been developed in the literature to aid novices or semi-experts in a variety of data management tasks. The focus of PBE is to not only solve the task, but also provide the *mechanism* that can solve the task. To this end, all PBE tools learn from the user examples and synthesize programs that can produce the desired results. To help data scientists write complex codes for data wrangling and data transformation, WREX [38] proposes an example-driven program synthesis approach. To enable integration of web data with spreadsheets, WebRelate [40] facilitates joining semi-structured web data with relational data in spreadsheets using input–output examples. FlashRelate [41] and FlashExtract [42] enable extraction of relational data from semi-structured spreadsheets, text files, and web pages, using examples. Data-transformation-by-example approaches [33,43] led to the development of the FlashFill [49] feature in Microsoft Excel, which can learn the user's data transformation intent only from a few examples. Beyond data management tasks, recently, PBE has been used for text processing [46], text normalization [47], and personalized text summarization [48]. Live programming [5] helps novice programmers to understand their codes, where they can manipulate the input by directly editing the codes and manipulate the output by providing examples of the desired output. Beyond computational tasks, PBE tools also support creative tasks such as music creation by example [50], where a software takes a song as an example and allows the user to interactively mix the AI-generated music.

Query by example (QBE) was an early effort to assist users without SQL expertise in formulating SQL queries [51]. Existing QBE systems [7,8] identify relevant relations and joins in situations where the user lacks schema understanding, but are limited to project-join queries. These systems focus on the common structure of the example tuples, and do not try to learn the common semantics as SQuID does. QPlain [9] uses user-provided provenance of the example tuples to learn the join paths and improve intent inference. However, this assumes that the user understands the schema, content, and domain to provide these provenance explanations, which is often unrealistic for nonexperts.

Set expansion is a problem corresponding to QBE in Knowledge Graphs [17,52,53]. SPARQLByE [54], built on top of a SPARQL QRE system [55], allows querying RDF datasets by annotated (positive/negative) example tuples. In semantic knowledge graphs, systems address the entity set expansion problem using maximal-aspect-based entity model, semantic-feature-based graph query, and entity co-occurrence information [11,18,19,56]. These approaches exploit the semantic context of the example tuples, but they cannot learn new semantic properties, such as aggregates involving numeric values, that are not explicitly stored in the knowledge graph, and they cannot express derived semantic properties without exploding the graph size. For example, to represent "appearing in more than K comedies", the knowledge graph would require one property for each possible value of K.

Interactive approaches rely on relevance feedback on system-generated tuples to improve query inference and result delivery [12–14,57,58]. Such systems typically expect a large number of interactions, and are often not suitable for nonexperts who may not be sufficiently familiar with the data to provide effective feedback.

Query reverse engineering (QRE) [59,60] is a special case of QBE that assumes that the provided examples comprise the complete

output of the intended query. Because of this closed-world assumption, QRE systems can build data classification models on denormalized tables [20], labeling the provided tuples as positive examples and the rest as negative. Such methods are not suitable for our setting, because we operate with few examples, under an open-world assumption. While few QRE approaches [29] relax the closed world assumption (known as the *superset QRE* problem) they are also limited to PJ queries similar to the existing QBE approaches. Most QRE methods are limited to narrow classes of queries, such as PJ [29,30], aggregation without joins [61], or top-k queries [62]. REGAL+[63] handles SPJA queries but only considers the schema of the example tuples to derive the joins and ignores other semantics. In contrast, SQuID considers joining relations without attributes in the example schema (Example 1.1). A few QRE methods do target expressive SPJ queries [15,16], but they only work for very small databases (<100 cells), and do not scale to the datasets used in our evaluation. Moreover, the user needs to specify the data in their entirety, thus expecting complete schema knowledge, while SCYTHE [15] also expects hints from the user towards precise discovery of the constants of the query predicates.

Alternative approaches exist, beyond by-example methods, to aid novice users explore relational databases. Keyword-based search [64–66] allows accessing relational data without knowledge of the schema and SQL syntax, but does not facilitate search by examples. Other notable systems that aim to assist novice users in data exploration and complex query formulation are: QueRIE, a query recommendation based on collaborative filtering [67], SnipSuggest, a context-aware SQL autocompletion system [68], SQL-Sugg, a keyword-based query suggestion system [69], YmalDB, a "you-may-also-like"-style data exploration system [70], and SnapToQuery, an exploratory query specification assistance tool [71]. These approaches focus on assisting users in query formulation, but assume that the users have sufficient knowledge about the schema and the data. VIDA [72], ShapeSearch [73], and Zenvisage [74] are visual query systems that allow visual data exploration, but they require the user to be aware of the trend within the output. Some approaches exploit user interaction to assist users in query formulation and result delivery [12–14,57,58]. There, the user has to provide relevance feedback on system-generated tuples. However, such highly interactive approaches are not suitable for data exploration as users often lack knowledge about the system-provided tuples, and thus, fail to provide correct feedback reflecting their query intent. Moreover, such systems often require a large number of user interactions. User-provided examples and interactions appear in other problem settings, such as learning schema mappings [75–77]. The query likelihood model in IR [78] resembles our technique, but does not exploit the similarity of the input entities.

Related work on user study of PBE approaches. Drosos et al. [38] present a comparative user study contrasting WREX against manual programming. The study results indicate that data scientists are more effective and efficient at data wrangling with WREX over manual programming. Mayer et al. [79] presents comparative study between two user interaction models – program navigation and conversational clarification – that can help resolve the ambiguities in the examples in by-example interaction models. Lee et al. [80] presents an online user study on how PBE systems help the users solve complex tasks. They identify seven types of mistakes commonly made by the users while using PBE systems, and also suggest an actionable feedback mechanism based on unsuccessful examples. Santolucito et al. [22] studied the impact of PBE on real-world users over a tool for shell scripting by example. Their study results indicate that while the users are quicker to solve the task using the PBE tool, they trust the traditional approach more. However, none of these studies focus on QBE in particular, which is a PBE system tailored towards data exploration over relational databases. The performance of a QBE tool is affected by additional factors, such as the subjectivity of the data exploration task and the domain knowledge of the user. Moreover, traditional data access and exploration methods pose hurdles not only to novices, but to expert users as well. These factors indicate the need for a new study that targets QBE systems in particular and are the motivation behind our comparative user studies.

10.2. Comparison with prior work

We provided a summary of prior work to contrast with SQuID in the comparison matrix of Fig. 3. We now explain the comparison metrics and highlight the key differences among different classes of query by example techniques and their variants. We organize the prior work into three categories: QBE (query by example), QRE (query reverse engineering), and DX (data exploration). Furthermore, we group QBE methods into two sub-categories, methods for relational databases, and methods for knowledge graphs. All QRE and DX methods that we discuss are developed on relational databases. We first describe our comparison metrics:

Query class encodes the expressivity of a query. We use four primitive SQL operators (join, projection, selection, and aggregation) as comparison metrics. While data retrieval mechanisms (e.g., graph query, SPARQL) for knowledge graphs do not directly support all these operators, they support similar expressivity through alternative equivalent operators.

Semi-join is a special type of join which is particularly useful for QBE systems. A system is considered to support semi-join if it allows inclusion of relations in the output query that have no attribute projected in the input schema (e.g., in Example 1.1, no attribute of `research` appears in the input tuples, but `Q2` includes `research`). While knowledge-graph-based systems do not directly support semi-join as defined in the relational database setting, they support same expressivity through alternative mechanism.

Implicit property refers to the properties that are not directly stated in the data (e.g., number of comedies an actor appears in). In SQuID, we compute implicit properties by aggregating direct properties of affiliated entities.

Scalability characterizes how the system scales when data increases. While deciding on scalability of a system, we mark a system scalable only if it either had a rigorous scalability experiment, or was shown to perform well on real-world big datasets. Thus, we do not consider approaches as scalable if the dataset is too small (e.g., contains 100 cells).

Open-world assumption states that what is not known to be true is simply unknown. In QBE and related work, if a system assumes that tuples that are not in the examples are not necessarily outside of user interest, it follows the open-world assumption. In contrast, closed-world assumption states that when a tuple is not specified in the user example, it is definitely outside of user interest.

Apart from the aforementioned metrics, we also report any *additional requirement* of each prior art. We briefly discuss different types of additional requirements here: *User feedback* involves answering any sort of system generated questions. It ranges from simply providing relevance feedback (yes/no) to a system-suggested tuple to answering complicated questions such as "if the input database is changed in a certain way, would the output table change in this way?" Another form of requirement involves providing *negative tuples* along with positive tuples. *Provenance input* requires the user to explain the reason why they provided each example. Some systems require the user to provide the example tuples sorted in a particular order (top-k), aiming towards reverse engineering top-k queries. *Schema-knowledge* is assumed when the user is supposed to provide provenance of examples or sample input database along with the example tuples.

Comparison summary. QBE methods on relational databases largely focus on project-join queries. Few knowledge-graph-based approaches support attribute value specification, which is analogous to selection predicates in relational databases. However, they are limited to predicates involving categorical attributes or simple comparison operators

(= and ≠) involving numerical attributes. This is a serious practical limitation as user intent is often encoded by range predicates on numeric attributes. Therefore, we mark such limited support with gray circles.

While all QBE methods follow open-world assumption, QRE methods are usually built on the closed-world assumption. However, few QRE methods also support open-world assumption and support superset QRE variation. However, such approaches are limited to PJ queries only. In general, QRE methods cannot support highly expressive class of queries without severely compromising scalability.

While almost every QBE and QRE technique supports join and projection, data exploration techniques usually assume that the tuples reside in a denormalized table and the entire rows of relevant entities are of user interest; thus, data exploration techniques do not focus on deriving the correct join path or projection columns.

Contrast with machine learning. Machine learning methods can model QBE settings as classification problems, and relational machine learning targets relational settings in particular [81]. However, while the provided examples serve as positive labels, QBE settings do not provide explicit negative examples. Semi-supervised statistical relational learning techniques [82] can learn from unlabeled and labeled data, but require unbiased sample of negative examples. There is no straightforward way to obtain such a sample in our problem setting without significant user effort.

Our problem setting is better handled by one-class classification [83,84], more specifically, Positive and Unlabeled (PU) learning [21,85–89], which learns from positive examples and unlabeled data in a semi-supervised setting [90]. Most PU-learning methods assume denormalized data, but relational PU-leaning methods do exist. However, all PU-learning methods rely on one or more strong assumptions [87] (e.g., all unlabeled entities are negative [91], examples are selected completely at random [21,92], positive and negative entities are naturally separable [85,86,93], similar entities are likely from the same class [94]). These assumptions create a poor fit for our problem setting where the example set is very small, it may exhibit user biases, response should be real-time, and intents may involve deep semantic similarity.

Existing PU-learning approaches over relational data make some strong assumptions that do not fit into our problem setting. Under the *SCAR* assumption, TIcER [92] estimates label frequency, which is the sampling rate of examples, to solve the PU-learning problem over relational data. However, when the number of positive examples is small, it generates high-precision, but low-recall classifier. Under the *separability* assumption, few PU-learning approaches [85,86] infer reliable negative examples from the positive examples and apply iterative learning to converge to the final classifier, which is prohibitive for the real-time data exploration setting. Aleph [93] is a relational rule learning system that allows a `PosOnly` setting for PU-learning, based on the separability assumption. However, it tries to minimize the size of the retrieved data, which results in low-recall with very few examples. Under the *smoothness* assumption, RelOCC [94] uses positive examples and exploits the paths that the examples take within the underlying relational data to learn distance measure. However, it does not exploit any aggregated feature (deep semantic similarity) or feature statistics (selectivity) obtained from the entire dataset. We summarize the key points to contrast machine learning (ML) approaches with SQuID below:

Dependency on data volume. SQuID is agnostic to the volume of unlabeled data as it relies on highly compressed summary of the feature statistics (e.g., selectivity of the filters), precomputed over the data. SQuID pushes this summarization task in the offline preprocessing step and uses the summary during online intent discovery. In contrast, efficiency of ML approaches depends on the sheer volume of the data as they are data-intensive. Sampling is a way to deal with large volume of data, however, it comes at a cost of information loss and reduced accuracy. Moreover, for large data spread out in diverse classes, it is hard to produce an unbiased sample; it is even harder to produce such sample in a relational dataset. Ideally, ML approaches are task-specific and the large training time is affordable due to being a one-time requirement. However, a query intent discovery system is designed for data exploration, which demands real-time performance. Each query intent is equivalent to a new machine-learning task and requires time-consuming training, which is not ideal in the data exploration setting.

Training effort. For each task, ML approaches require training a new model, which requires significant effort (e.g., manual hyperparameter tuning) to converge to a model. Therefore, ML approaches would need to rebuild the model every time a new query intent is posed, or even when the current example set is augmented with new examples. No single hyperparameter setting would work for all tasks where the tasks are unknown a-priori. Under the separability assumption, some PU-learning approaches [85] apply iterative learning to converge to the final classifier which is wasteful for learning each query intent. In contrast, SQuID does not require hyperparameter tuning for each task, rather it only requires one-time manual parameter tuning for the overall intent types (e.g., user preference regarding precision–recall tradeoff) on a particular dataset.

Interpretability. SQuID is a query by example method which is an instantiation of general programming by example (PBE). One key difference between PBE and ML is the requirement of interpretability of the underlying model. The goal of PBE technique is to provide the users the learned model (e.g., SQL query in our case), not just a black box that separates the intended data from the unintended one. In contrast, the focus of ML approaches is to construct a model, often extremely complex (e.g., deep neural network), that separates the positive data from the negative ones.

Functioning with very few examples. Even though PU-learning approaches work with only positive examples, they require a fairly large fraction of the positive data as examples. In contrast, SQuID works on very small set of examples which is natural for data exploration. This is possible under the strong assumption that the underlying model, where the user examples are sampled from, is a structured query. This implies that the user consistently provides semantically similar examples reflecting their true intent. When the labeled data is this small, PU-learning approaches result in high-precision, but low-recall classifiers, which does not help in data exploration.

Assumptions involving model and feature prior. One significant distinction between SQuID and ML approaches is the assumption regarding the underlying model. SQuID assumes that there exists a SQL query with conjunctive selection predicates (features) that is capable to generate the complete set of positive tuples. In contrast, ML approaches do not have any such simplified assumption and attempts to learn a separating criteria based on features. Hence, it is unlikely for ML systems to drop strongly correlated features observed within the examples, despite being coincidental. Additionally, we exploit two information—(1) data dependent feature prior (Section 4.2.1), and (2) data-independent feature prior (Section 4.2.2)—which is hard to incorporate in ML. As an example, in Section 4.2.2, we discuss outlier impact, a non-trivial component of feature prior, which indicates whether a set of features together is likely to be intended. Such assumptions cannot be encoded in ML approaches in a straightforward way.

Contrast with data cube. Data cube [95] can serve as an alternative mechanism to model the information precomputed in the abduction-ready database αDB. However, a principal contribution of the αDB is the determination of which information is needed for SQuID's inference, rendering it much more efficient than a data cube solution. We have empirically evaluated data cube's performance on the IMDb database using Microsoft SQL Server Analysis Services (SSAS), where we defined a three-dimensional data cube: (`person`, `movie`, `genre`), deployed it in Microsoft Analysis Server 14 with process option "Process

Full", and used MDX queries to extract data. We also ported the relevant SQuID αDB data (`persontogenre`) from PostgreSQL into Microsoft SQL Server 14, and evaluated the corresponding SQL queries. We found that the data cube performs *one to two orders of magnitude slower* than queries over the αDB. One can materialize certain summary-views by applying roll-up operations on the data cube to expedite query execution, but such materializations essentially replicate the information materialized in the αDB; and determining the appropriate data to materialize, i.e., which derived relations to precompute, is the primary contribution of the αDB.

If one were to materialize all possible roll-up operations to take advantage of data cube's generality without the performance penalty, this would require *four orders of magnitude* more space compared to the αDB on the IMDb data. Compression mechanisms exist to store sparse data cubes efficiently, but such compression would hurt the query performance even more. The issue here is that the data cube encodes meaningless views (e.g., `person-to-movie`), because genre is not an independent dimension with respect to movie. In contrast, SQuID aggregates out large entity dimensions (e.g., SQuID aggregates out `movie` while computing `persontogenre`) which ensures that the size of the αDB is reasonable (Fig. A.29). Ultimately, even though the data cube does provide a possible mechanism for encoding the αDB data, it is not well-suited for schemas that do not exhibit the independence of dimensions that the data cube inherently assumes, resulting in poor performance compared to the αDB.

11. Conclusions

In this article, we focused on the problem of query intent discovery from a set of example tuples. We presented SQuID, a system that performs query intent discovery effectively and efficiently, even with few examples in most cases. The insights of our work rely on exploiting the rich information present in the data to discover similarities among the provided examples, and distinguish between those that are coincidental and those that are intended. Our contributions include a probabilistic abduction model and the design of an abduction-ready database, which allow SQuID to capture both explicit and implicit semantic contexts. Our work includes an extensive experimental evaluation of the effectiveness and efficiency of our framework over three real-world datasets, case studies based on real user-generated examples and abstract intents, and comparison with the state-of-the-art query reverse engineering technique and with PU-learning. Our empirical results highlight the flexibility of our method, as it is extremely effective in a broad range of scenarios. Notably, even though SQuID targets query intent discovery with a small set of a examples, it outperforms the state of the art in query reverse engineering in most cases, and is superior to learning techniques.

Our comparative user studies found that database users, with varied levels of prior SQL expertise, are significantly more effective and efficient at a variety of data exploration tasks with SQuID over the traditional SQL querying mechanism that requires database schema understanding and manual programming. Our results indicate that SQuID eliminates the barriers of familiarizing oneself with the database schema, formally expressing the semantics of an intended task, and writing syntactically correct SQL queries. The key takeaway of this work is that in a programming-by-example tool like SQuID, even a limited level of domain expertise (knowledge of a subset of the desired data) can substantially help overcome the lack of technical expertise (knowledge of SQL and schema) in data exploration and retrieval. This indicates that programming by example can lead to the democratization of complex computational systems and make these systems accessible to novice users while aiding expert users as well. Our studies validate some prior results over other PBE approaches but also contribute new empirical insights and suggest future directions for QBE systems to further increase system explainability and user trust.

CRediT authorship contribution statement

Anna Fariha: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Lucy Cousins:** Writing – original draft, Visualization, Software, Methodology. **Narges Mahyar:** Methodology. **Alexandra Meliou:** Supervision, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Alexandra Meliou reports financial support was provided by National Science Foundation. Anna Fariha reports a relationship with Microsoft Corporation that includes: employment and equity or stocks. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by National Science Foundation, United States under the grants IIS-2453461 and CNS-2346555, a Stena Center for Financial Technology Center Seed Grant, a One Utah Data Science Hub Seed Grant, a Google DANI research award, and project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the European Union under the NextGenerationEU Program.

Appendix. Datasets and benchmark queries

We collected the datasets from various sources and we provide them in Fig. A.28. We provide the detailed description of the datasets in Figs. A.29 and A.30. We mention the cardinalities of the large relations for providing a sense of the data size and associations among relations.

Title	Source
IMDb dataset	https://datasets.imdbws.com/
DBLP dataset	https://data.mendeley.com/datasets/3p9w84t5mr
Adult dataset	https://archive.ics.uci.edu/ml/datasets/adult
Physically strong actors	https://www.imdb.com/list/ls050159844/
Top 1000 Actors and Actresses*	http://www.imdb.com/list/ls058011111/
Sci-Fi Cinema in the 2000s	http://www.imdb.com/list/ls000097375/
Funny Actors	https://www.imdb.com/list/ls000025701/
100 Random Comedy Actors	https://www.imdb.com/list/ls000791012/
BEST COMEDY ACTORS	https://www.imdb.com/list/ls000076773/
115 funniest actors	https://www.imdb.com/list/ls051583078/
Top 35 Male Comedy Actors	https://www.imdb.com/list/ls006081748/
Top 25 Funniest Actors Alive	https://www.imdb.com/list/ls056878567/
the top funniest actors in hollywood today	https://www.imdb.com/list/ls007041954/
Google knowledge graph: Actors: Comedy	https://www.google.com/search?q=funny+actors
The Best Movies of All Time*	https://www.ranker.com/crowdranked-list/the-best-movies-of-all-time
Top H-Index for Computer Science & Electronics*	http://www.guide2research.com/scientists/

Fig. A.28. Source of datasets and lists used in this article. * denotes the lists that are used as popularity mask.

IMDb & variations				
IMDb			bd-IMDb	
	DB size	633 MB	DB size	1926 MB
	#Relations	15	#Relations	15
	Precomputed DB size	2310 MB	Precomputed DB size	5971 MB
	Precomputation time	150 min	Precomputation time	370 min
Relation Cardinality	person	6,150,949	person	12,301,898
	movie	976,719	movie	1,953,438
	castinfo	14,915,325	castinfo	59,661,300
bs-IMDb			sm-IMDb	
	DB size	1330 MB	DB size	75 MB
	#Relations	15	#Relations	15
	Precomputed DB size	4831 MB	Precomputed DB size	317 MB
	Precomputation time	351 min	Precomputation time	14 min
Relation Cardinality	person	12,301,898	person	65,865
	movie	1,953,438	movie	335,705
	castinfo	29,830,650	castinfo	1,364,890

Fig. A.29. Description of different variations of the IMDb dataset.

	DBLP		Adult	
	DB size	22 MB	DB size	4 MB
	#Relations	14	#Relations	1
	Precomputed DB size	98 MB	Precomputed DB size	5 MB
	Precomputation time	42 min	Precomputation time	3 min
Relation Cardinality	author	126,094		
	publication	148,521	adult	32,561
	authortopub	416,445		

Fig. A.30. Description of the DBLP and the Adult datasets.

ID	Task	J	S	#Result
IQ1	Entire cast of Pulp Fiction	3	1	113
IQ2	Actors who appeared in all of The Lord of the Rings trilogy	8	7	20
IQ3	Canadian actresses born after 1970	3	4	1531
IQ4	Sci-Fi movies released in USA in 2016	5	3	1374
IQ5	Movies Tom Cruise and Nicole Kidman acted together	5	2	12
IQ6	Movies directed by Clint Eastwood	4	2	36
IQ7	All movie genres	1	0	35
IQ8	Movies by Al Pacino	4	2	71
IQ9*	Indian actors who acted in at least 15 Hollywood movies	6	4	23
IQ10*	Actors who acted in more than 10 Russian movies after 2010	6	4	84
IQ11	Hollywood Horror-Drama movies in 2005 – 2008	7	5	291
IQ12	Movies produced by Walt Disney Pictures	3	1	394
IQ13	Animation movies produced by Pixar	5	2	57
IQ14	Sci-Fi movies acted by Patrick Stewart	6	3	22
IQ15	Japanese Animation movies	5	2	2512
IQ16*	Walt Disney Pictures movies with more than 15 American cast members	5	3	207

* Includes `GROUP BY` and `HAVING` clauses

Fig. A.31. Benchmark queries for the IMDb dataset. **J** and **S** denote the number of joins and selection predicates, respectively.

A.1. Alternative IMDb datasets

For the scalability experiment, we generated 3 versions of the IMDb database (Fig. A.29). For obtaining a downsized database `sm-IMDb`, we dropped persons with less than 2 affiliated movies and/or who have too many semantic information missing, and movies that have no cast information. We produced two upsized databases: one with dense associations `bd-IMDb`, and the other with sparse associations `bs-IMDb`. The database `bd-IMDb` contains duplicate entries for all movies, persons, and companies (with different primary keys), and the associations among persons and movies are duplicated to produce more dense associations. For example, if person `P1` appeared in movie `M1` in

ID	Task	J	S	#Result
DQ1	Authors who collaborated with both U Washington and Microsoft Research Redmond	5	2	30
DQ2*	Authors with at least 10 SIGMOD and at least 10 VLDB publications	8	4	52
DQ3	SIGMOD publications in 2010 – 2012	3	3	468
DQ4	Publications Jiawei Han, Xifeng Yan, and Philip S. Yu published together	7	3	15
DQ5	Publications between USA and Canada	5	2	336

* Includes `GROUP BY`, `HAVING`, and `INTERSECT`

Fig. A.32. Benchmark queries for the DBLP dataset. **J** and **S** denote the number of joins and selection predicates, respectively.

SQL Query	#Result
`SELECT DISTINCT name FROM adult WHERE education = 'Bachelors' AND occupation = 'Craft-repair' AND hoursperweek >= 36 AND hoursperweek <= 40 AND age >= 46 AND age <= 47`	8
`SELECT DISTINCT name FROM adult WHERE race = 'White' AND sex = 'Female' AND nativecountry = 'United-States' AND relationship = 'Other-relative' AND occupation = 'Machine-op-inspct' AND workclass = 'Private'`	11
`SELECT DISTINCT name FROM adult WHERE occupation = 'Craft-repair' AND workclass = 'Private' AND age >= 65 AND age <= 68 AND relationship = 'Husband'`	12
`SELECT DISTINCT name FROM adult WHERE maritalstatus = 'Divorced' AND capitalgain >= 7298 AND capitalgain <= 10520 AND hoursperweek >= 40 AND hoursperweek <= 44 AND relationship = 'Not-in-family'`	14
`SELECT DISTINCT name FROM adult WHERE capitalgain >= 4101 AND capitalgain <= 4650 AND workclass = 'Private' AND age >= 41 AND age <= 44`	14
`SELECT DISTINCT name FROM adult WHERE occupation = 'Protective-serv' AND hoursperweek >= 45 AND hoursperweek <= 48`	44
`SELECT DISTINCT name FROM adult WHERE education = '10th' AND race = 'White' AND fnlwgt >= 334113 AND fnlwgt <= 403468`	48
`SELECT DISTINCT name FROM adult WHERE nativecountry = 'United-States' AND hoursperweek >= 43 AND hoursperweek <= 45 AND race = 'White' AND fnlwgt >= 106541 AND fnlwgt <= 118876`	126
`SELECT DISTINCT name FROM adult WHERE race = 'White' AND education = 'Bachelors' AND nativecountry = 'United-States' AND capitalgain >= 6097 AND capitalgain <= 7688 AND maritalstatus = 'Married-civ-spouse' AND relationship = 'Husband'`	128
`SELECT DISTINCT name FROM adult WHERE education = 'Bachelors' AND capitalloss >= 1848 AND capitalloss <= 1980`	182
`SELECT DISTINCT name FROM adult WHERE sex = 'Male' AND nativecountry = 'United-States' AND capitalloss >= 1848 AND capitalloss <= 1887`	203
`SELECT DISTINCT name FROM adult WHERE education = 'Doctorate' AND maritalstatus = 'Married-civ-spouse' AND nativecountry = 'United-States'`	223
`SELECT DISTINCT name FROM adult WHERE education = 'HS-grad' AND workclass = 'Private' AND hoursperweek >= 45 AND hoursperweek <= 46 AND relationship = 'Husband'`	241
`SELECT DISTINCT name FROM adult WHERE capitalgain >= 7688 AND capitalgain <= 8614`	343
`SELECT DISTINCT name FROM adult WHERE education = 'Bachelors' AND maritalstatus = 'Never-married' AND workclass = 'Private' AND hoursperweek >= 40 AND hoursperweek <= 43 AND race = 'White'`	563
`SELECT DISTINCT name FROM adult WHERE education = 'HS-grad' AND nativecountry = 'United-States' AND occupation = 'Machine-op-inspct' AND race = 'White'`	777
`SELECT DISTINCT name FROM adult WHERE nativecountry = 'United-States' AND age >= 60 AND age <= 62`	798
`SELECT DISTINCT name FROM adult WHERE fnlwgt >= 271962 AND fnlwgt <= 288781`	912
`SELECT DISTINCT name FROM adult WHERE maritalstatus = 'Married-civ-spouse' AND fnlwgt >= 221366 AND fnlwgt <= 259301`	1340
`SELECT DISTINCT name FROM adult WHERE maritalstatus = 'Never-married' AND fnlwgt >= 185624 AND fnlwgt <= 211177`	1404

Fig. A.33. Benchmark queries for the Adult dataset.

IMDb, i.e., (P1,M1) exists in IMDb's castinfo, we added a duplicate person P2, a duplicate movie M2, and 3 new associations, (P1,M2), (P2,M2), and (P2,M1), to bd−IMDb's castinfo. For bs−IMDb, we only duplicated the old associations, i.e., we added P2 and M2 in a similar manner, but only added (P2,M2) in castinfo.

A.2. Benchmark queries

Figs. A.31 and A.32 show benchmark queries that we used to run different experiments on the IMDb and the DBLP datasets, respectively. The tables show the query intents, details of the corresponding queries in SQL (number of joining relations (J) and selection predicates (S)), and the result set cardinality. Fig. A.33 shows 20 benchmark queries along with their result set cardinality for the Adult dataset.

Data availability

Data will be made available on request.

References

[1] A. Parameswaran, Democratizing data science and lessons learned along the way, in: PVLDB PhD Workshop, 2020.
[2] Y. Pu, K. Ellis, M. Kryven, J. Tenenbaum, A. Solar-Lezama, Program synthesis with pragmatic communication, 2020, arXiv prhideeprint arXiv:2007.05060.
[3] A. Cypher, Eager: programming repetitive tasks by example, Readings in Human–Computer Interaction, Elsevier, 1995, pp. 804–810.
[4] H. Lieberman, Programming by example (introduction), Commun. ACM 43 (3) (2000) 72–74.
[5] M. Santolucito, W.T. Hallahan, R. Piskac, Live programming by example, in: CHI, 2019.
[6] S. Gulwani, Programming by examples: applications, algorithms, and ambiguity resolution, in: IJCAR, 2016, pp. 9–14.
[7] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, L. Novik, Discovering queries based on example tuples, in: SIGMOD, 2014, pp. 493–504.
[8] F. Psallidas, B. Ding, K. Chakrabarti, S. Chaudhuri, S4: Top-k spreadsheet-style search for query discovery, in: SIGMOD, 2015, pp. 2001–2016.
[9] D. Deutch, A. Gilad, Qplain: query by explanation, in: ICDE, 2016, pp. 1358–1361.
[10] D. Mottin, M. Lissandrini, Y. Velegrakis, T. Palpanas, Exemplar queries: a new way of searching, VLDBJ 25 (6) (2016) 741–765.
[11] L. Lim, H. Wang, M. Wang, Semantic queries by example, in: EDBT, 2013, pp. 347–358.
[12] A. Bonifati, R. Ciucanu, S. Staworko, Learning join queries from user examples, TODS 40 (4) (2016) 24:1–24:38.
[13] H. Li, C. Chan, D. Maier, Query from examples: An iterative, data-driven approach to query construction, PVLDB 8 (13) (2015) 2158–2169.
[14] K. Dimitriadou, O. Papaemmanouil, Y. Diao, AIDE: an active learning-based approach for interactive data exploration, TKDE 28 (11) (2016) 2842–2856.
[15] C. Wang, A. Cheung, R. Bodik, Synthesizing highly expressive sql queries from input–output examples, in: PLDI, 2017, pp. 452–466.
[16] S. Zhang, Y. Sun, Automatically synthesizing sql queries from input–output examples, in: ASE, 2013, pp. 224–234.
[17] X. Zhang, Y. Chen, J. Chen, X. Du, K. Wang, J. Wen, Entity set expansion via knowledge graphs, in: SIGIR, 2017, pp. 1101–1104.
[18] S. Metzger, R. Schenkel, M. Sydow, QBEES: query-by-example entity search in semantic knowledge graphs based on maximal aspects, diversity-awareness and relaxation, J. Intell. Inf. Syst. 49 (3) (2017) 333–366.
[19] N. Jayaram, A. Khan, C. Li, X. Yan, R. Elmasri, Querying knowledge graphs by example entity tuples, TKDE 27 (10) (2015) 2797–2811.
[20] Q.T. Tran, C. Chan, S. Parthasarathy, Query reverse engineering, VLDBJ 23 (5) (2014) 721–746.
[21] C. Elkan, K. Noto, Learning classifiers from only positive and unlabeled data, in: SIGKDD, 2008, pp. 213–220.
[22] M. Santolucito, D. Goldman, A. Weseley, R. Piskac, Programming by example: efficient, but not "helpful", in: PLATEAU@SPLASH, 2018, pp. 3:1–3:10.
[23] T. Menzies, Applications of abduction: knowledge-level modelling, Int. J. Hum. -Comput. Stud. 45 (3) (1996) 305–335.
[24] A.C. Kakas, Abduction, in: Encyclopedia of Machine Learning and Data Mining, Springer, 2017, pp. 1–8.
[25] L.E. Bertossi, B. Salimi, Causes for query answers from databases: Datalog abduction, view-updates, and integrity constraints, Int. J. Approx. Reasoning 90 (2017) 226–252.
[26] O. Arieli, M. Denecker, B.V. Nuffelen, M. Bruynooghe., Coherent integration of databases by abductive logic programming, J. Artif. Intell. Res. 21 (2004) 245–286.
[27] M. Dredze, D. Rao P. McNamee, A. Gerber, T. Finin, et al., Entity disambiguation for knowledge base population, in: Proceedings of the 23rd International Conference on Computational Linguistics, 2010, pp. 277–285.
[28] S. Agarwal, A. Sureka, N. Mittal, R. Katyal, D. Correa, DBLP records and entries for key computer science conferences, 2016.
[29] D.V. Kalashnikov, L.V. S. Lakshmanan, D. Srivastava, FastQRE: fast query reverse engineering, in: SIGMOD, 2018, pp. 337–350.
[30] M. Zhang, H. Elmeleegy, C.M. Procopiuc, D. Srivastava, Reverse engineering complex join queries, in: SIGMOD, 2013, pp. 809–820.
[31] S. Liu, Most used languages among software developers globally 2020, 2020.
[32] C. Baik, Z. Jin, M.J. Cafarella, H.V. Jagadish, Duoquest: A dual-specification system for expressive SQL queries, in: SIGMOD, 2020, pp. 2319–2329.
[33] S. Gulwani, Automating string processing in spreadsheets using input–output examples, in: POPL, 2011, pp. 317–330.
[34] A. Fariha, S.M. Sarwar, A. Meliou, Squid: semantic similarity-aware query intent discovery, in: G. Das, C.M. Jermaine, P.A. Bernstein (Eds.), Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, ACM, 2018, pp. 1745–1748.
[35] A. Fariha, A. Meliou, Example-driven query intent discovery: abductive reasoning using semantic similarity, PVLDB 12 (11) (2019) 1262–1275.
[36] S. Jha, S. Gulwani, S.A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, ICSE, vol. 1, IEEE, 2010, pp. 215–224.
[37] M. Raza, S. Gulwani, Disjunctive program synthesis: A robust approach to programming by example, in: AAAI, 2018, pp. 1403–1412.
[38] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, Sumit Gulwani, Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists, in: CHI'20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25–30, 2020, ACM, 2020, pp. 1–12, http://dx.doi.org/10.1145/3313831.3376442.
[39] S. Gulwani, Programming by examples - and its applications in data wrangling, in: Dependable Software Systems Engineering, 2016, pp. 137–158.
[40] J.P. Inala, R. Singh, Webrelate: integrating web data with spreadsheets using examples, POPL 2 (POPL) (2017) 1–28.
[41] D.W. Barowy, S. Gulwani, T. Hart, B.G. Zorn, FlashRelate: extracting relational data from semi-structured spreadsheets using examples, in: PLDI, 2015, pp. 218–228.
[42] V. Le, S. Gulwani, FlashExtract: a framework for data extraction by examples, in: PLDI, 2014, pp. 542–553.
[43] Y. He, K. Ganjam, K. Lee, Y. Wang, V.R. Narasayya, S. Chaudhuri, X. Chu, Y. Zheng, Transform-data-by-example (TDE): extensible data transformation in excel, in: SIGMOD, 2018, pp. 1785–1788.
[44] X. Wang, S. Gulwani, R. Singh, FIDEX: filtering spreadsheet data using examples, in: OOPSLA, 2016, pp. 195–213.
[45] J.K. Feser, S. Chaudhuri, I. Dillig, Synthesizing data structure transformations from input–output examples, ACM SIGPLAN Not. 50 (6) (2015) 229–239.
[46] K. Yessenov, S. Tulsiani, A.K. Menon, R.C. Miller, S. Gulwani, B.W. Lampson, A. Kalai, A colorful approach to text processing by example, in: UIST, 2013, pp. 495–504.
[47] D. Kini, S. Gulwani, FlashNormalize: programming by examples for text normalization, in: IJCAI, 2015, pp. 776–783.
[48] A. Fariha, M. Brucato, A. Meliou, P.J. Haas, SuDocu: Summarizing documents by example, in: PVLDB, 2020.
[49] FlashFill. https://tinyurl.com/flashfillmicrosoft.
[50] E. Frid, C. Gomes, Z. Jin, Music creation by example, in: CHI, 2020, pp. 1–13.
[51] M.M. Zloof, Query-by-example: the invocation and definition of tables and forms, in: PVLDB, 1975, pp. 1–24.
[52] R.C. Wang, W.W. Cohen, Language-independent set expansion of named entities using the web, in: ICDM, 2007, pp. 342–350.
[53] Word grabbag, 2018, http://wordgrabbag.com.
[54] G.I. Diaz, M. Arenas, M. Benedikt, SPARQLByE: Querying RDF data by example, PVLDB 9 (13) (2016) 1533–1536.
[55] M. Arenas, G.I. Diaz, E.V. Kostylev, Reverse engineering SPARQL queries, in: WWW, 2016, pp. 239–249.
[56] J. Han, K. Zheng, A. Sun, S. Shang, J.R. Wen, Discovering neighborhood pattern queries by sample answers in knowledge base, in: ICDE, 2016, pp. 1014–1025.
[57] A. Abouzied, D. Angluin, C. Papadimitriou, J.M. Hellerstein, A. Silberschatz., Learning and verifying quantified boolean queries by example, in: PODS, 2013, pp. 49–60.
[58] X. Ge, Y. Xue, Z. Luo, M.A. Sharaf, P.K. Chrysanthis, Request: a scalable framework for interactive construction of exploratory queries, in: Big Data, 2016, pp. 646–655.
[59] Y.Y. Weiss, S. Cohen, Reverse engineering spj-queries from examples, in: PODS, 2017, pp. 151–166.
[60] P. Barceló, M. Romero., The complexity of reverse engineering problems for conjunctive queries, in: ICDT, vol. 68, 2017, pp. 7:1–7:17.
[61] W.C. Tan, M. Zhang, H. Elmeleegy, D. Srivastava, Reverse engineering aggregation queries, PVLDB 10 (11) (2017) 1394–1405.

[62] K. Panev, N. Weisenauer, S. Michel, Reverse engineering top-k join queries, in: BTW, 2017, pp. 61–80.
[63] W.C. Tan, M. Zhang, H. Elmeleegy, D. Srivastava, REGAL+: reverse engineering SPJA queries, PVLDB 11 (12) (2018) 1982–1985.
[64] S. Agrawal, S. Chaudhuri, G. Das., DBXplorer: a system for keyword-based search over relational databases, in: ICDE, 2002, pp. 5–16.
[65] V. Hristidis, Y. Papakonstantinou, DISCOVER: keyword search in relational databases, in: VLDB, 2002, pp. 670–681.
[66] Z. Zeng, M. Lee, T.W. Ling, Answering keyword queries involving aggregates and GROUPBY on relational databases, in: EDBT, 2016, pp. 161–172.
[67] M. Eirinaki, S. Abraham, N. Polyzotis, N. Shaikh, Querie: collaborative database exploration, TKDE 26 (7) (2014) 1778–1790.
[68] N. Khoussainova, Y. Kwon, M. Balazinska, D. Suciu, SnipSuggest: context-aware autocompletion for SQL, 4, (1) 2010, pp. 22–33,
[69] J. Fan, G. Li, L. Zhou, Interactive sql query suggestion: making databases user-friendly, in: ICDE, 2011, pp. 351–362.
[70] M. Drosou, E. Pitoura, Ymaldb: exploring relational databases via result-driven recommendations, VLDBJ 22 (6) (2013) 849–874.
[71] L. Jiang, A. Nandi, SnapToQuery: providing interactive feedback during exploratory query specification, PVLDB 8 (11) (2015) 1250–1261.
[72] D.J. L. Lee, A.G. Parameswaran, The case for a visual discovery assistant: A holistic solution for accelerating visual data exploration, IEEE Data Eng. Bull. 41 (3) (2018) 3–14.
[73] T. Siddiqui, P. Luh, Z. Wang, K. Karahalios, A.G. Parameswaran, Shapesearch: A flexible and efficient system for shape-based exploration of trendlines, in: SIGMOD, 2020, pp. 51–65.
[74] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, A.G. Parameswaran, Effortless data exploration with zenvisage: an expressive and interactive visual analytics system, PVLDB 10 (4) (2016) 457–468.
[75] A.D. Sarma, A.G. Parameswaran, H. Garcia-Molina, J. Widom, Synthesizing view definitions from data, in: ICDT, 2010, pp. 89–103.
[76] L. Qian, M.J. Cafarella, H.V. Jagadish, Sample-driven schema mapping, in: SIGMOD, 2012, pp. 73–84.
[77] A. Bonifati, U. Comignani, E. Coquery, R. Thion, Interactive mapping specification with exemplar tuples, in: SIGMOD, 2017, pp. 667–682.
[78] C.D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008.
[79] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B.G. Zorn, S. Gulwani, User interaction models for disambiguation in programming by example, in: UIST, 2015, pp. 291–301.
[80] T.Y. Lee, C. Dugan, B.B. Bederson, Towards understanding human mistakes of programming by example: an online user study, in: IUI, 2017, pp. 257–261.
[81] L. Getoor, B. Taskar, Introduction to statistical relational learning, 2007.
[82] R. Xiang, J. Neville, Pseudolikelihood EM for within-network relational learning, in: ICDM, 2008, pp. 1103–1108.
[83] L.M. Manevitz, M. Yousef, One-class svms for document classification, 2, (Dec) 2001, pp. 139–154,
[84] S.S. Khan, M.G. Madden, A survey of recent trends in one class classification, in: AICS, Springer, 2009, pp. 188–197.
[85] H. Yu, J. Han, K.C. Chang, PEBL: positive example based learning for web page classification using SVM, in: SIGKDD, 2002, pp. 239–248.
[86] B. Liu, Y. Dai, X. Li, W.S. Lee, S.Y. Philip, Building text classifiers using positive and unlabeled examples, in: ICDM, vol. 3, Citeseer, 2003, pp. 179–188.
[87] J. Bekker, J. Davis, Learning from positive and unlabeled data: A survey, 2018, CoRR, arXiv:abs/1811.04820.
[88] J. Bekker, J. Davis, Estimating the class prior in positive and unlabeled data through decision tree induction, in: AAAI, 2018, pp. 2712–2719.
[89] F. Mordelet, J.-P. Vert, A bagging svm to learn from positive and unlabeled examples, Pattern Recognit. Lett. 37 (2014) 201–209.
[90] O. Chapelle, B. Schlkopf, A. Zien, Semi-Supervised Learning, 1st edn, The MIT Press, 2010.
[91] A. Neelakantan, B. Roth, A. McCallum, Compositional vector space models for knowledge base completion, in: ACL, 2015, pp. 156–166.
[92] J. Bekker, J. Davis, Positive and unlabeled relational classification through label frequency estimation, in: ILP, 2017, pp. 16–30.
[93] A. Srinivasan, The aleph manual.
[94] T. Khot, S. Natarajan, J.W. Shavlik, Relational one-class classification: A non-parametric approach, in: AAAI, 2014, pp. 2453–2459.
[95] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals, Data Min. Knowl. Discov. 1 (1) (1997) 29–53.