Interactively Discovering Query from Example and Explanation

Anna Fariha
University of Massachusetts Amherst
afariha@cs.umass.edu

Sheikh Muhammad Sarwar
University of Massachusetts Amherst
smsarwar@cs.umass.edu

ABSTRACT

Database management systems are heavily used for storing structured data and retrieving relevant data as per user requirement. However, a large group of database users are not expert in query language and they lack knowledge about database schema. As a result, they struggle in cases where data retrieval requires them to formulate complex SQL queries with complete knowledge about database schema. Interestingly, the users are often aware of a few example tuples and the properties or concepts of the majority of the values present in those tuples that enable them to explain their information need. However, to the best of our knowledge, there exists no framework that can help the user to communicate those information to a database system. In this project, we propose *QBEE - Query by Example and Explanation* framework, that takes a few example tuples and the associated concepts for each of the attribute values in those tuples, and finally translates those into a corresponding SQL query. The resultant SQL query can produce the result set desired by the user. In order to achieve this, the system interacts with the user to collect the explanations behind the user provided example tuples. So far, we have been successful to generate concepts given a single tuple, present those concepts to the user using a text-based interface, and finally construct the user's intended query using her feedback on concepts on IMDB dataset.

1 INTRODUCTION

As size of database and complexity in database schema increase, it becomes harder for the users to retrieve information of their interest by writing complex SQL queries. As a result, keyword based data acquisition becomes essential for relational databases. Often database users are not completely aware of the complex underlying schema or lack expertise on query language, and they fail to retrieve the data tuples of their interest. Since a large group of non-expert database users require to retrieve data from database, it becomes a challenge for them when it comes to formulate queries, often with multiple joins with complex constraints. This requires complete knowledge of the underlying database schema as well as knowledge about query language. Fortunately, users are often aware of values of few attributes of the tuples that are sample of the data of their interest. Moreover, they are also aware of "why" that sample represents the data of interest.

Our goal is to reverse engineer the query of user's intention from the *examples* given by the user along with the associated *explanations*. For a database $\mathcal{D}$, and a set of example tuples (E) as well as their explanations (X_E) given by a user, there can be a set of candidate queries ($\mathcal{Q} = Q_1, Q_2, ..., Q_n$), where $\forall Q_i \in \mathcal{Q}, E \subseteq Q_i(\mathcal{D})$. One of the most reasonable ways to find the user's target query Q_i, is to provide a suitable interface to the user that can enable her to provide explanations. Explanations help a system to understand why the example tuples provided by the user qualify her information need.

Given a set of example tuples, E, some challenges are: (1) how to "capture" the associated explanations X_E, (2) how to find $\mathcal{Q}$ efficiently from E and (3) how to construct the query $Q' \in \mathcal{Q}$ reflecting user's actual intention with the help of X_E. The task to discover query from example is challenging to accomplish without any subsequent user feedback for eliminating some of the candidate queries. It is even more challenging to do so without understanding the reason behind the choice of example and subsequent feedback. We propose a framework that leverages user provided explanations X_E associated with E towards retrieving the query that "captures" E as well as the user's intention. To achieve this goal, we incorporate the notion of *Concept* based explanation, associated with values of example tuples, aiming to understand user's intention. The user provided explanation will help to narrow down search space by eliminating false positives queries.

1.1 Motivating Example

We provide a motivating example in this section. Consider the database in Table 1. The database contains three relations: `Flight Origin` shown in Table 1(a), `Flight Status` shown in Table 1(b) and `Weather` shown in Table 1(c). Also note that there is an example tuple in Table 1(d). From this example, a number of different valid candidate queries can be found. A few of those are shown in Table 2. All of these candidate queries are valid in a sense that they "capture" the user provided example tuple. In order to find the actual user intention and consequently, the result of the query that the user possesses in her mind, we require explanations from the user regarding why she put each attribute value in the example tuple.

For example, we can ask the user "Why `Boston`?", and the answer could be "Because it `Snowed` in Boston on Feb-18". To answer the question "Why flight `SP-977`?", user might say that "Because the flight was `Delayed`". These two explanations lead us to choose Q3 in Table 2(c) as the most qualified query. The user might not provide explanation for some attributes meaning that she wants exact match for that particular value. For example in Table 2(b), the user is simply interested in *all* flights originating from `Boston`, possibly just because she is interested in flights leaving from `Boston`. Moreover, in this case, she chose flight `SP-977`, just because it is one of the flights she knows about that originated from `Boston`. Looking at Q1 in Table 2(a), here the user is not concerned about the originating city, rather about the status `Delayed` of the flight. So the query goal is to discover `Delayed` flights originating from *any* city.

To answer questions in cases like Q1-Q3, there exists an approach – QPlain [8, 9] that enables the users to provide explanations by selecting intermediate tuples from the database that contribute to the example tuple. This approach is suitable for cases where a join

City	Date	Flight
Hartford	Feb-16	AA-354
Boston	Feb-16	UA-001
Boston	Feb-18	SP-977
Boston	Feb-18	GP-718
Chicago	Feb-19	MA-887
Chicago	Feb-19	DA-004
Portland	Feb-12	AK-234
Portland	Feb-12	SD-236

(a) Relation: Flight Origin

Flight	Status
AA-354	On-time
UA-001	Delayed
SP-977	Delayed
GP-718	On-time
MA-887	Delayed
DA-004	Cancelled
AK-234	Delayed
SD-236	Cancelled

(b) Relation: Flight Status

City	Date	Weather
Hartford	Feb-16	Sunny
Boston	Feb-16	Sunny
Boston	Feb-18	Snow
Chicago	Feb-19	Snow
Portland	Feb-12	Rain

(c) Relation: Weather

Boston	Feb-18	SP-977

(d) Example tuple

Table 1: Motivating example

City	Date	Flight	Status
Boston	Feb-16	UA-001	Delayed
Boston	Feb-18	SP-977	Delayed
Chicago	Feb-19	MA-887	Delayed
Portland	Feb-12	AK-234	Delayed

(a) Q_1: Find all flights where flight `Status` = `Delayed`

City	Date	Flight	Status
Boston	Feb-16	UA-001	Delayed
Boston	Feb-18	SP-977	Delayed
Boston	Feb-18	GP-718	On-time

(b) Q_2: Find all flights where origin `City` = `Boston`

City	Date	Flight	Status
Boston	Feb-18	SP-977	Delayed
Chicago	Feb-19	MA-887	Delayed

(c) Q_3: Find all flights where flight `Status` = `Delayed` and origin city's `Weather` = `Snow`

City	Date	Flight	Status
Boston	Feb-18	SP-977	Delayed
Chicago	Feb-19	MA-887	Delayed
Chicago	Feb-19	DA-004	Cancelled
Portland	Feb-12	AK-234	Delayed
Portland	Feb-12	SD-236	Cancelled

(d) Q_4: Find all flights where origin city's `Weather` ∈ {`Snow`, `Rain`} or flight's `Status` ∈ {`Delayed`, `Cancelled`}

Table 2: Result for different candidate queries

path can be constructed from the corresponding explanation tuples for each example tuple. However, this approach requires the user to fully understand how a database join works and she has to be precise while selecting the contributing explanatory tuples. This contradicts with the initial motivation behind the work, that query from example is a scheme to aid non-expert database users to formulate query from example tuples. Consider the case where the user has Q4, shown in Table 2(d), in her mind. She is particularly interested about flight interruptions due to some extreme weather condition in the originating city. When QPlain will require the user to select explanatory tuples to explain the example tuple in Table 1(d), the user will provide the same explanations as she will do for Q3, that is, `Boston` is covered by `Snow` and flight `SP-977` is `Delayed`. However, to provide additional explanations for extreme weather conditions such as `Rain` or `Tornedo` and other flight anomalies such as `Cancelled`, she has to come up with corresponding representative examples. If she is unable to provide sufficient example tuples to indicate all possible explanations representing the query intention in her mind, QPlain would not be able to find her intended query.

Motivated by these shortcomings of the existing approach and to alleviate the user's effort to explain the example tuple, we use the notion of `Concept`. Users can choose from a set of concepts associated with the example attribute values to provide more precise explanation. To explain our approach, let us consider that the user has Q4 (Table 2(d)) in her mind. When she explains `Boston`, we provide her with the concepts related to the constant `Boston` (ellipses with solid border). Additionally, we also provide them related concepts (ellipses with dashed border). For example, when the user is asked "why `Boston`", she can select one (or many) concepts associated with `Boston`. She can either choose `Snow`, or she can choose another relevant concept, e.g.m `Rain`.

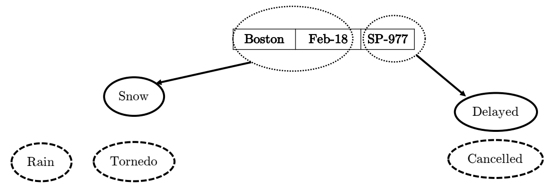

Figure 1: Concept mapping for explanation

Similarly while explaining `flight SP-977`, she can now choose the concept `Cancelled`, along with `Delayed`, where the latter was the only option using QPlain. The main advantage of using concepts in providing explanation is, the user can browse related concepts and pick all the concepts that are in her mind. She can also pick some concepts of those she did not think of initially. Figure 1 reflects the idea to explain the example tuple in Table 1(d) when Q4 in Table 2(d) is the actual user intention. Here, `Boston` is mapped to the concept `Snow` and some related concepts are also available. As a result, any originating city with one of the selected extreme weather condition will qualify as a desired flight's originating city. Similarly, flight `SP-977` is mapped to the concept `Delayed`, but additional related concepts are also proved to the user. Observe that the tuple `(Portland, Feb-12, SD-236)` has nothing in common with the example tuple, if the explanation was `Snow` for `Boston` and `Delayed` for `SP-977`, because this flight was `Cancelled` due to `Rain` in `Portland` on that day. As we leverage related concepts, the user becomes able to provide additional explanations for the example tuples in a more natural way.

1.2 Motivating Example from IMDB

In this section, we provide another motivating example from IMDB [1] dataset. Let us consider that a user is interested to find movies that casted a pair of Oscar winner actors. To find such movies, she gives an example tuple `(Leonardo DiCaprio, Kate Winslet, Titanic)`. Even though the user is looking for Oscar winner actor pairs, this example can also indicate an American-British actor pair. It can also indicate an Oscar winner movie, for which none of the actors won the Oscar. We can also have multiple concepts associated with a single attribute. The user might look for an American Oscar winner actor and British actress pair. Then, for `Leonardo DiCaprio`, we have three different concepts: `American`, `Oscar recipient`, and `Male`, whereas for `Kate Winslet` we have one concept: `British`. Overall, we can conclude that without getting clarification about the concepts through feedback from the user, it is nearly impossible to find her desired query.

2 LITERATURE SURVEY

To aid non-expert database users in data exploration and data acquisition, prior research works have been done in several directions. Keyword based search on relational database mostly retrieves minimal set of tuples (possibly through joins among multiple relations) containing the keywords of interest. Another line of research aims towards helping users in generating, reusing and formulating query statements for data retrieval. Data driven query discovery has been extensively visited, where users do not require to deal with the complexity of query language syntax. Interactivity has been introduced in all of these lines of works which can further improve the non-expert users' experience in accessing databases. Few approaches leverage provenance of the example data for finding queries from those given examples. In this section, we briefly discuss these research paths and show how they contrast with our vision towards using explanations and user interaction for finding data given few examples.

2.1 Query Recommendation, Auto-completion and Reusing

There have been prior works to help non-expert database users to facilitate formulating queries or obtaining desired results. Works on query recommendation [7, 14] aimed to facilitate users with similar information need. These works leveraged past query logs and are completely based on previous events, ignoring the fact that current intentions of a user might differ from previous ones. Another work [13] provided "you may also like" style recommendation to the database users in a form of additional items assumed to be highly relevant to the user's information need. Khoussainova et al. [18] proposed a technique to interactively help user to choose predicates of SQL query by suggesting query fragments. There are other existing techniques [19] that enable users to reuse past queries by presenting them with the most common queries from user log data. All of these approaches mainly focus towards aiding users to formulate queries in existing query languages, and assume that users have sufficient knowledge about the underlying structure of the database.

2.2 Interactive Data Exploration

Prior works have been conducted [1, 17, 21] to interactively learn user's intention regarding expressing database query for data exploration. Abouzied et al. [1] proposed an interactivity based approach to learn whether system generated examples are *answer* or *non-answer* to the user's intended query. Dimitriadou et al. [10–12] developed an interactive data exploration framework (AIDE) aiming towards assisting users in data exploration without the need of formulating expensive exploratory queries. AIDE interactively collects user's feedback on relevance of sample data to their data exploration objective and finally generates a plausible query using classification technique. While AIDE learns to classify a data object as relevant or not from the attributes of the object, it does not aim to learn their relationships based on user preference.

Ge et al. [15] provided another interactive framework, REQUEST, to assist non-expert database users in query-from-example style data exploration while maintaining interactive speed. These interactive data exploration approaches aim to identify the data instances that is of user's interest by eliminating irrelevant data, but do not capture the relationship among the values of example tuples identified as relevant by the user.

Bonifati et al. [4, 5] proposed an interactive scheme to find out join queries and disjunctions from relations, where users interactively assign positive or negative label to tuples. Like some of the earlier works [30], they did not assume any referential integrity constraint, and only considered equijoin and semijoin queries putting additional constraints on the problem. However, this approach was also targeted towards identifying the query, not the data it generates.

2.3 Query from Example

Possibly the first graphical query language was query-by-example, devised by Zloof et al. [31], which was designed to reduce the user's burden by translating few user provided values to query statements. For search engines, Mottin et al. [22] introduced the term *exemplar query* to denote the notion of representing relationship among keywords of a search query (attributes of a tuple). The exemplar query acts as a sample from the expected result and used by the system to capture relationship among the attribute values.

Several sample driven schema mapping approaches [2, 3, 6, 25] have been proposed to discover the relationship to the schema from

[1]IMDB dataset: ftp://ftp.fu-berlin.de/pub/misc/movies/database/

user provided examples. One of the earliest works for reverse engineering queries in relational database was QBO (Query By Output) by Tran et al. [28], which was a data driven approach to produce instance-equivalent query (IEQ) Q' of the original query Q, given the database D and the query output $Q(D)$. The authors extended their work in [29] which could handle unknown input query Q using data classification based approach and optimization techniques. Zhang et al. [30] also proposed solution to the same problem for complex join queries that overcomes some constrains imposed on earlier works, but they are limited to project join queries only, lacking the support of selection conditions. Sarma et al. [26] also did similar work towards discovering view definition (unions of conjunctive queries) from view V and database D given as input and producing view definition Q as output assuming same schema for view V and single database relation R.

Panev et al. [23] worked on reverse engineering query given top-k result of that query, for conjunctive queries with equality predicates. There are additional works [24] that addressed the problem of finding approximate query from example tuples and incrementally produced results as the user provides values of the example tuples, in a spreadsheet style manner. Han et al. [16] put higher rank on the example tuples provided by the user to reverse engineer the query that produces ranked result as output.

Shen et al. [27] proposed algorithm that can solve the problem of constructing query based on only few example tuples, where tuples could be partially filled with possibly missing attribute values. This work aims to discover a set of valid queries (considering *project join* queries with *foreign key* join only), all of them satisfying the user provided example tuple set. Although they take into consideration that users may not be able to provide all attribute values for all example tuples, they do not provide scope for user to interact with the system that could possibly aid in the query discovering process. Additionally, they do not consider *selection*, but user's query intention can, in many cases, contain selection; and it is indeed hard to consider this class of queries without getting feedback from user interactively within the process. Another drawback of their work is that they present an unranked set of valid queries irrespective of user's actual intention.

Li et al. [20] solved the same problem of finding queries from example tuples, but in an interactive fashion. Given an initial database-result pair (D, R), they iteratively construct minimally modified database-result pair (D', R') such a way that it can be used to verify the correctness of a "guessed" query. This approach relies on user's feedback on a sequence of verifications with a divide and conquer strategy, and eventually prunes all false positive candidate queries. They adopted prior work [29] for finding candidate queries and supported SPJ classes of queries. Like [29, 30], this approach also required the example tuples to be completely filled. However, they require the user to provide initial database, which inherently require the user to be aware of the schema, and complete example tuples with no missing attribute values. Additionally, their interaction interface is too complex and time consuming for an ordinary non-expert user to respond on. The approach becomes intractable from user perspective when there are lots of relations involved in the query inferring process.

2.4 Query from Provenance

The approaches described in 2.3 require the users to come up with plenty of initial examples that are representative of the expected output to succeed. However, coming up with a lot of representative examples are sometimes difficult and with few examples, it is hard to capture the true intention of an user from the examples they provide. The interesting observation here is, although users might not know a lot of examples but only a few, they are usually aware of the underlying reason of why they are providing the examples. This observation can help us leverage the concept of data provenance for discovering queries from example tuples. A very recent line of work [8, 9] takes data provenance into consideration in the query-from-example problem. However, they are limited in the way users can provide explanation as they only support users to select relevant tuples that serve as the provenance information. The drawback of their approach is: for providing explanation, users have to somewhat know apriori the complex database schema and also which tuples contribute to the examples she provided. This can be a tedious job assuming the users we are talking about are non-expert users. Additionally, for each join path, that the user wishes to be exploited while producing result, the user has to provide a representative example and corresponding join path as explanation. For disjunctions on two alternatives, this will require exponential number of example queries to represent each join path.

3 PROPOSED APPROACH

Our objective is significantly different from the approaches that focus on finding a set of valid queries from example, because our target is to find a single query of user intention. This query will generate a result set that best describes the relationships among the attributes in the user provided example tuples. Hence, it is of prime importance to get user feedback for binding the concepts with each attribute of her given tuples, and use those concepts in turn to discover attribute relationships. Overall, QBEE does not require users to possess much prior knowledge about the data or the database schema and only a single example tuple is enough at the beginning. The proposed framework performs concept fixing and concept binding for leveraging user provided explanation to find the target query. In Section 3.1 - 3.6 we provide preliminary definitions and use those to describe our proposed approach in Section 3.7.

3.1 Preliminaries

QBEE relies on the assumption that the database it is working on is normalized using BCNF. This assumption helps us to define and use our terms precisely.

Definition 3.1. *Objects* are entities in the database and *Concepts* are properties of objects. Example of objects in the context of IMDB dataset are person `Leonardo DiCaprio`, movie `Titanic` and production company `Disney`. A user can make use of these objects to form an example tuple by providing attribute values of that object, for example, name for a person. Objects of same type are referred to as *Object Class*. Object class defines the required attributes of its instances, i.e., objects. In IMDB database `title` attribute of the object class `Movie` contains title of a movie and a user can provide title of a specific movie (e.g., `Titanic`) within the example tuple.

Definition 3.2. *Concepts* are the properties of objects. Given an object `Kate Winslet`, there exists several associated concepts such as `Female`, `British`, `Singer` etc. that can be associated with it. *Concept Attributes* are the attributes that holds the concepts. For example, the concepts attribute that contains the concept `Female` is `Gender`.

Definition 3.3. *Dimension Table* describes objects and concepts. The schema of dimension table is derived from object classes and concept attributes. Basically dimension table contains data, mostly textual, that completely describes objects and concepts. We denote *Object Dimension Tables* as `DO` and *Concept Dimension Tables* as `DC`. Examples of object dimension tables in Table 3 are 3(a) and 3(b), whereas example of concept dimension table is 3(e).

Definition 3.4. *Fact Table* contains relationships among objects and concepts. Basically fact table consists of foreign keys to dimension tables. We denote *Object Concept Fact Tables* as `FOC` and *Object Object Fact Table* as `FOO`. Example of object concept fact table in Table 3 is 3(d) and example of object object fact table is 3(c).

Definition 3.5. *Basic Concept Attribute* can be found by the following join paths. $DO \bowtie FOC \bowtie DC$. Here, joins are limited to foreign key joins only. Intuitively, basic concepts denote the properties of object itself.

Definition 3.6. *Derived Concept Attribute* can be found by the following join paths. $DO \bowtie FOO \bowtie FOC \bowtie DC$. Here, joins are limited to foreign key joins only. Intuitively, derived concepts denote the properties of a related object of an object.

Definition 3.7. Given an object attribute, the task of *Concept Mapping* refers to the process of finding all possible concept attributes of that object.

Example 3.1. *Let us assume that in the context of IMDB, a user initiated our system with example tuple:* (`Leonardo Dicaprio`, `Kate Winslet`, `Titanic`). `Leonardo Dicaprio` *appears in the* `name` *attribute of the dimension table* `Person` *in the database as shown in Table 3(a).* `Gender` *is a basic concept attribute of the object of object class* `Person` *with name* `Leonardo Dicaprio`. *We can have another derived concept attribute* `Genre`, *from the sample database shown in Table 3, which can be associated with* `Person`. *To acquire this derived concept attribute, we would have to perform the following join:* `Person` ⋈ `CastInfo` ⋈ `MovieGenre` ⋈ `Genre`. *However, this relationship is many-to-many and thus it is very hard to associate an actor with a specific genre as shown in Table 3(f). We take a frequency-based approach and keep only the top-k frequent concepts associated with each object. For example, if* `k = 1`, *we will only associate the concept* `Drama` *with* `Leonardo DiCaprio` *as it has maximum* `Count = 2` *in Table 3(f). Note that, in some cases, there might not exist any semantic relationship between a derived concept and an object.*

Example 3.2. *Let us assume that the attribute value* `Leonardo Dicaprio` *of our example tuple is affiliated with* `Romantic`, `Thriller` *and* `Action` *concepts for derived concept attribute* `Genre`; *the user selects the concept* `Thriller` *as an explanation. However, the user might also be interested in other genres like* `Horror`, *that might not be affiliated with* `Leonardo Dicaprio` *at all. But, we can not disregard it as the user might really be looking for an actor who acted in a thriller or horror movie. So, QBEE provides options to further explore a selected concept attribute so that the user can look for an actor who has done* `Thriller` *or* `Horror` *movies. At the time of producing user intended query QBEE applies disjunction over all the selected concepts. The SQL query constructed for this example is given below:*

```
  Select Name from Person, CastInfo, MovieGenre, Genre
where [FK joining conditions] and (Genre.Genre='Thriller'
or Genre.Genre='Horror')
```

PersonID	Name	Gender
1	Kate Winslet	female
2	Leonardo Dicaprio	male
3	Jim Carrey	male

(a) Relation: Person

MovieID	Name	Year
1	Titanic	2001
2	Blood Diamond	2005

(b) Relation: Movie

MovieID	PersonID
1	1
1	2
2	2

(c) Relation: CastInfo

MovieID	GenreID
1	1
1	2
2	1
2	3

(d) Relation: MovieGenre

GenreID	Genre
1	Drama
2	Romantic
3	Thriller
4	Horror

(e) Relation: Genre

PersonID	GenreID	Count
2	1	2
2	2	1
2	3	1

(f) Relation: Derived Concept Attribute

Table 3: Sample IMDB database

3.2 Concept Fixing

Given the example tuple, we fix concept for each of the object values in the tuple. As the concepts reside in table columns i.e. concept attributes, we take a single concept, pair it up with the corresponding concept attribute, and present them to the user. This is helpful in cases where attribute name carries no semantic meaning. In the `Leonardo DiCaprio` example, we present several concepts such as `USA` (for `Nationality`), `Thriller` (for `Genre` of his movies), `Male` (for `Gender`) etc. If a user selects `Thriller`, then we deduce that she is interested in the concept attribute `Genre` of an actor, and then we prompt the user for concept binding.

3.3 Concept Binding

After the user has fixed a concept attribute, we present her the possible values for that concept attribute associated with the given example. In our example, we show the three concepts – `Romantic`, `Thriller` and `Action` associated with `Leonardo Dicaprio` if the concept attribute fixed in the previous phase was `Genre`. The user then specifies the genres of her interest.

3.4 Concept Exploration

In the concept binding phase, the user might not find all the concepts she is interested in. As for example, if we only show the genres associated with `Leonardo DiCaprio`, it might not cover all the genres the user wants to specify. Even though the user has given an example, it does not necessarily cover her point of interest. So, along with associated concepts, we also show the most frequently occurring concepts for the fixed concept attribute to the user for exploring the concept space. After this phase, we can get all the concepts a user is interested against a concept attribute. We apply disjunction of them to produce the predicate for this concept attribute. The steps of concept fixing, binding and exploration is repeated several times until the user is done providing explanation for different concept attributes.

3.5 Concept Ranking

There can be a number of concepts associated with a concept attribute. Given an object `Leonardo DiCaprio`, several genre concepts can be attached with it, as a person can act in different types of movies. So, concepts need to be ranked before they are presented to the user. Although we have not completed this implementation, but we will use frequency based concept ranking, i.e., we will present the genre of movies in which `Leonardo DiCaprio` acted the maximum number of times as the top genre concept.

3.6 Global Inverted Column Index

Global Inverted Column Index [27] is used to look up the table name and column name that contains a text value. Table 4 shows an excerpt of the global inverted column index that has been constructed using the relations presented in Table 3.

Text	Table Name	Attribute Name
Kate Winslet	Person	Name
Titanic	Movie	Name
Horror	Genre	Genre

Table 4: Relation: Global Inverted Column Index

3.7 QBEE Algorithm

The goal of QBEE is to find out the query that is actually in user's mind given the example tuples and associated explanations. The key steps of the proposed QBEE algorithm is provided below:

(1) User provides example tuples in her desired schema.
(2) Candidate relation-attribute pairs are identified using global inverted column index to generate candidate object classes for each of the columns provided in example schema.
(3) To disambiguate between multiple candidate objects, concept attributes can be used in the concept fixing phase. For example, if there are two candidate object classes `(Person, Movie)` for an example column containing value `Madonna`, concept fixing on concept attribute `Gender` disambiguates between the candidate object classes and narrows it down to only `Person`.
(4) Concept fixing is also used to narrow down user's point of interest in terms of concept attributes.
(5) Concept binding and concept exploration phases are used to collect concepts that falls within user's interest.
(6) Steps 3–5 is repeated until the user terminates providing explanation.
(7) A project-join query is constructed with foreign key joins among dimension and fact tables that satisfies the user provided example schema after object class disambiguation.
(8) With the explanation collected from user regarding the example tuples, predicates are added to the constructed project-join query to construct a select-project-join query. QBEE supports conjunction among predicates for multiple concept attributes and disjunction among concepts within a single concept attribute.
(9) The constructed select-project-join query is executed and the results of that query is shown to the user. This result should contain all the example tuples provided by the user.

In the next section, we discuss about the implementation of the method and provide experimental results.

4 EVALUATION

In this section, at first we describe our dataset by detailing about the schema, dataset size, index and concept attributes. Then we discuss about our project implementation environment. Finally, we discuss about the evaluation strategy and present the experimental results.

4.1 Dataset

We have used IMDB dataset for the experiments. We have created a new schema that is in BCNF and imported data from raw dataset to our implemented schema. Next we briefly discuss about the schema and distinguish among different types of tables defined in 3.3 and 3.4 within the schema.

4.1.1 Schema. Figure 2 represents the schema diagram that we used to run our experiments. The leftmost group in pink color shows four concept dimension tables, each denoting concept attributes related to object class `movie`. The yellow group denotes four object-concept fact tables. These tables express relationships among `movie` objects and affiliated concepts. The red group represents three object dimension tables – `movie`, `person` and `company`. The green group shows three object-object fact tables representing relationships among pair of objects – `castinfo` (`movie`, `person`), `distribution` (`movie`, `company`) and `production` (`movie`, `company`). Finally, the blue group contains only one table, and it denotes the relationship concept attribute that is affiliated with the relationship `castinfo`.

Note that, the tables are indexed using several columns. Other than the primary keys, object dimension tables are indexed using

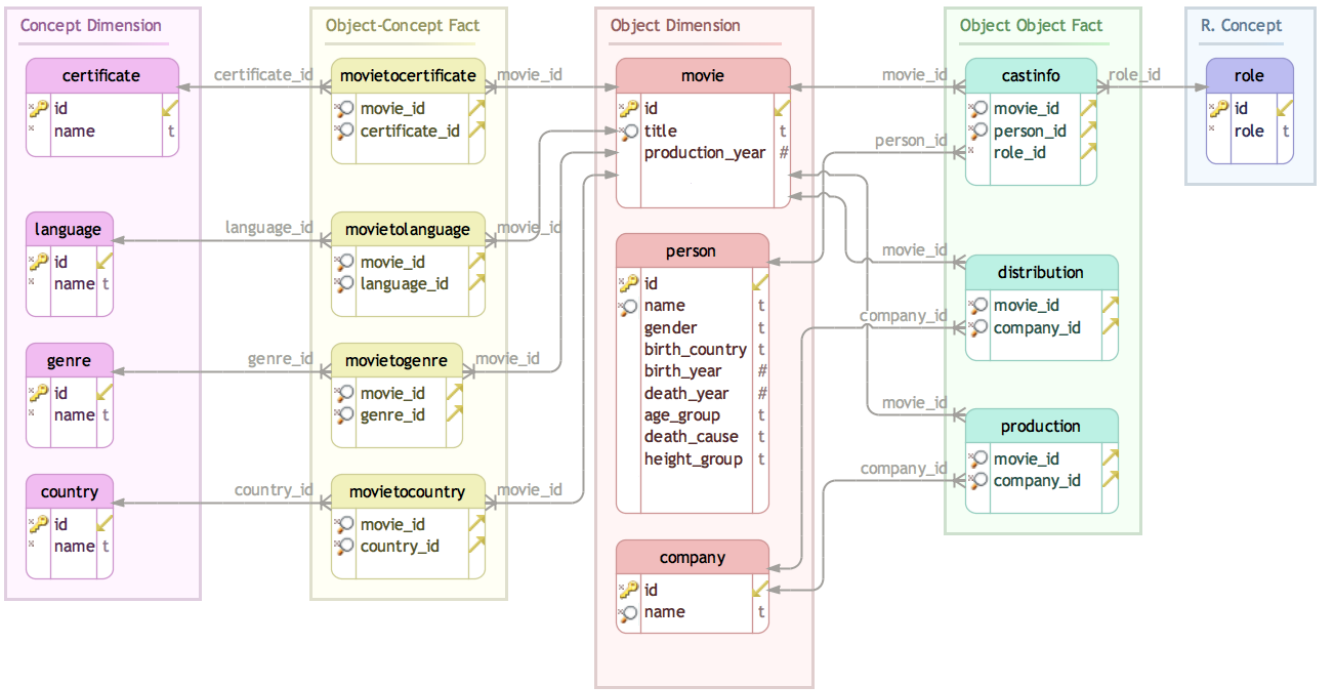

Figure 2: Schema for IMDB dataset

the most probable search columns (e.g., `movie.title`, `person.name`, `company.name`). The fact tables are also indexed by appropriate foreign keys (e.g., `castinfo.movie_id`, `movietogenre.movie_id`, `movietogenre.genre_id`). We have also clustered the tables based

Table	#Rows	Concept Attribute	#Concepts	Time(s)
movie	965,574	production_year	142	0.416
		certificate	491	0.004
		language	346	0.004
		genre	35	0.001
		country	242	0.003
person	6,053,766	gender	3	2.8
		birth_country	844	2.2
		birth_year	444	2.1
		death_year	394	2.1
		death_cause	341	2.2
		age_group	11	3.3
		height_group	9	2.4
company	338,527	N/A	N/A	N/A
inv. col. index	6,118,768	N/A	N/A	N/A

Table 5: Database size and concept retrieval time

on appropriate indexes that will expedite the query evaluation according to the user's need (e.g., `movietogenre.genre_id`). A special table, not shown in the schema, is the inverted column index table. This table is indexed using the composite key (`tab_name`, `col_name`) for quick retrieval of qualifying table and column names of values of the given example tuple.

4.1.2 *Dataset Size and Concept Retrieval Speed.*

Size of the database and table indexes are the two most important factors for determining the time required for retrieving concepts for presenting to the user. The expectation is that this will be done in an interactive speed. Table 5 presents the number of rows for each object dimension table, number of distinct concepts for each affiliated concept attribute and time required to present those concepts to the user. Number of rows for inverted column index is also shown here.

4.1.3 *Concept Attributes.*

In IMDB dataset, we found some useful concept attributes associated with object classes – `movie` and `person`. Here we provide small description of concept attributes associated with object class `movie`.

(1) `Production Year`: Numeric. Denotes the year of production of a movie. e.g., `2014`.
(2) `Certificate`: Categorical. Denotes the movie's certificates. It has a country associated. e.g., `USA:PG`.
(3) `Language`: Categorical. Denotes the languages the movie is released on. e.g., `French`.
(4) `Genre`: Categorical. Denotes the genres of the movie. e.g., `Horror`.
(5) `Country`: Categorical, Denotes the release countries for a movie. e.g., `Korea`.

We have also found a number of concept attributes with object class `person` and those are summarized in Table 5.

		User 1: Query (without QBEE)				User 2: Query (using QBEE)			
#	Task Description	Construction Time (min)	#Intermediate Queries	#Result Rows	Execution Time (sec)	Construction Time (min)	#User Interactions	#Result Rows	Execution Time (sec)
1	Find all (actor, movie) pairs where the actor is from `India` and the movie's genre is `Sci-fi` or `Adventure` and country is `USA`.	20	6	277	24.6	6	8	277	17.2
2	Find all (actor, movie) pairs where actor's birth country is `USA` and the movie has been produced in `USA` but banned in `South Korea`.	18	3	259	0.064	5	8	259	0.031
3	Find all (actor, movie) pairs, where movie was released in language `French` and actor is from `USA` and `dead`	15	3	9520	11	6	8	9520	6.9
4	Find all the `Romance` movies acted by `Tom Cruise`	2	1	14	1.4	2	6	11	1.4
5	Find all the movies directed by `Clint Eastwood` between `2000` and `2010`	4	3	10	1.4	2	15	10	1.4

Table 6: Evaluation summary on 5 different tasks

4.2 Evaluation Results

Now we discuss the performance of our system. We first discuss the implementation and evaluation environment followed by the the performance analysis.

4.2.1 Project Implementation Environment. We have used `Java` as the programming language and `PostgreSQL` as the database management system for implementing this project. As a first step, we have implemented a command line based application. We have future plan to enhance it by supporting graphical user interface. The system was built and tested in a machine with 2.7 GHz Intel Core I-5 processor with 8 GB of RAM and Mac OS 10.12.4.

4.2.2 Performance Analysis. To measure the performance of the designed system, we have generated 5 benchmark data retrieval tasks related to IMDB dataset. These tasks are expressed in a natural language. Two real persons are assigned to retrieve data from the database according to the task specifications with different system environments as described below.

User 1: The first user was provided with the database schema and a traditional SQL interface (postgresql) where he can see the schema, browse the data and execute any SQL query. We have measured the amount of time required to complete each of the assigned tasks as the user generates a SQL query for task accomplishment. We have also measured the time required by the DBMS to execute the user derived SQL query.

User 2: The second user was provided with the system QBEE. In this case, she has to come up with an example tuple on her own. To find a suitable example tuple, she was allowed to use an Internet search engine. We have measured the amount of time required in this case by adding up the time to come up with an example, time spent by interacting with the system and time required by the system to generate the SQL query. Finally, we measured the time required by the DBMS to execute QBEE generated SQL query.

Table 6 summarizes our findings. It includes the task description, query construction time by both users, number of intermediate queries or user interaction needed by the users, number of result rows returned by the final query and the amount of time required for the query execution. In all cases, the execution time required by QBEE generated query was less than the one that was generated by the first user. This is because QBEE optimizes the query by performing less number of joins. For example the first task required 8 joins for query generated by the first user, whereas qbee generated an equivalent query with 5 joins only. This is possible due to excluding few concept dimension tables and using the primary keys of required concepts instead.

Also note that, for task #4, the number of result rows are not equal for both users. We investigated the case manually later and found out that user 1 did a mistake by not including the role `actor` with `Tom Cruise`, which resulted in including few unwanted tuples where `Tom Cruise` directed a `Romance` movie. This phenomenon is less likely to occur with QBEE, as it will show the relationship attribute `role` to the user through the interface. It indicates that QBEE is also useful for getting rid of common errors. Another observation is, for task #5, where range query is necessary, QBEE requires a lot of interactions as it requires the user to specify each of the 11 years (`2000 - 2010`) manually. This is one shortcoming of QBEE and we plan to overcome this by incorporating range queries in future.

4.2.3 Discussion. After the user evaluation we discussed about their experiences. Both the users were efficient in forming SQL for data retrieval. The first user faced a number of problems those appear when writing complex SQL queries with many joins. Specifically, he was writing wrong join conditions that slowed down the process of finding the accurate SQL. As our tasks required the user to keep track of many predicates as well as join conditions, in one occasion the first user missed one predicate that delayed the process of finding the desired result set. The user also had to look at the attribute values of different attributes by using multiple projection queries. Additionally, he had to check few data types to write proper equality predicates. Occasionally, there were spelling mistakes in writing predicate values. However, as the tasks cover a small portion of IMDB data, the user quickly familiarized himself with the tables and join conditions, and efficiently completed the last two tasks, each taking less than 4 minutes. Overall, the user agreed that this traditional process was very inefficient for the first couple of tasks.

In our discussion with the second user, we found several positive feedback. QBEE did not require any effort to write complex join queries. Some of our tasks required joining of up to 8 tables, which was time consuming, and QBEE could reduce the effort significantly even with a text based interface. The only difficulty for the second user was to come up with an example that represents her information need. The second user also mentioned that incorporating range queries would significantly increase the usefulness of the system. Because, otherwise for a concept attribute such as `year`, she had to select several concepts to indicate a range of years.

The time and effort required to solve tasks for the second user did not vary much from query to query. Moreover, each of the tasks required only around 6 minutes for the second user. Comparing with the time taken for the first query of the first user, which is around 20 minutes, this is significantly low. The second user also pointed that switching to a GUI based interface would be a nice thing to have.

5 CONCLUSION AND FUTURE SCOPE

In this project, we designed and developed a system – QBEE, that helps database users to retrieve data without having detailed knowledge about the schema or SQL. The system incorporates user provided examples and subsequent feedback in the form of explanations for the examples. One of our major contributions is to devise a concept based approach to take user explanations and use them inside selection predicates to generate SQL queries automatically. Overall, the final goal of the system is to provide the user a well-formed SQL query that, when executed, will generate the data of user's need.

Prior approaches to solve our problem were limited in a sense that they could only generate project-join queries. QBEE shows enhanced capability by handling select-project-join classes of queries. We have presented the usefulness of our project by experimenting on real users. Apart from the ease that the user is facilitated with using QBEE, it also generates optimized query that a normal user usually does not come up with.

As a part of future works, we plan to investigate how we can efficiently compute derived concept attributes for objects. We also have plan to develop an automatic query inference engine that will initially come up with most probable user query even without any explanation. Another direction we plan to explore is concept ranking, which will aid the user to easily locate relevant concepts. Finally, we plan to apply diversification of result tuples to maximize user satisfaction.

REFERENCES

[1] Abouzied, A., Angluin, D., Papadimitriou, C., Hellerstein, J. M., and Silberschatz, A. (2013). Learning and verifying quantified boolean queries by example. In *Proceedings of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, pages 49–60, New York, NY, USA. ACM.

[2] Alexe, B., Chiticariu, L., Miller, R. J., and Tan, W.-C. (2008). Muse: Mapping understanding and design by example. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 10–19, Washington, DC, USA. IEEE Computer Society.

[3] Alexe, B., ten Cate, B., Kolaitis, P. G., and Tan, W.-C. (2011). Designing and refining schema mappings via data examples. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 133–144, New York, NY, USA. ACM.

[4] Bonifati, A., Ciucanu, R., and Staworko, S. (2014). Interactive inference of join queries. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 451–462.

[5] Bonifati, A., Ciucanu, R., and Staworko, S. (2016). Learning join queries from user examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38.

[6] Cate, B. T., Dalmau, V., and Kolaitis, P. G. (2013). Learning schema mappings. *ACM Trans. Database Syst.*, 38(4):28:1–28:31.

[7] Chatzopoulou, G., Eirinaki, M., Koshy, S., Mittal, S., Polyzotis, N., and Varman, J. S. V. (2011). The querie system for personalized query recommendations. *IEEE Data Eng. Bull.*, 34(2):55–60.

[8] Deutch, D. and Gilad, A. (2016a). Qplain: Query by explanation. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1358–1361.

[9] Deutch, D. and Gilad, A. (2016b). Query by provenance. *CoRR*, abs/1602.03819.

[10] Dimitriadou, K., Papaemmanouil, O., and Diao, Y. (2014a). Explore-by-example: An automatic query steering framework for interactive data exploration. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 517–528, New York, NY, USA. ACM.

[11] Dimitriadou, K., Papaemmanouil, O., and Diao, Y. (2014b). Interactive data exploration based on user relevance feedback. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 292–295.

[12] Dimitriadou, K., Papaemmanouil, O., and Diao, Y. (2016). Aide: An active learning-based approach for interactive data exploration. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2842–2856.

[13] Drosou, M. and Pitoura, E. (2013). Ymaldb: Exploring relational databases via result-driven recommendations. *The VLDB Journal*, 22(6):849–874.

[14] Eirinaki, M., Abraham, S., Polyzotis, N., and Shaikh, N. (2014). Querie: Collaborative database exploration. *IEEE Trans. Knowl. Data Eng.*, 26(7):1778–1790.

[15] Ge, X., Xue, Y., Luo, Z., Sharaf, M. A., and Chrysanthis, P. K. (2016). Request: A scalable framework for interactive construction of exploratory queries. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 646–655.

[16] Han, J., Zheng, K., Sun, A., Shang, S., and Wen, J. R. (2016). Discovering neighborhood pattern queries by sample answers in knowledge base. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1014–1025.

[17] Jiang, L. and Nandi, A. (2015). Snaptoquery: Providing interactive feedback during exploratory query specification. *Proc. VLDB Endow.*, 8(11):1250–1261.

[18] Khoussainova, N., Kwon, Y., Balazinska, M., and Suciu, D. (2010). Snipsuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33.

[19] Khoussainova, N., Kwon, Y., Liao, W., Balazinska, M., Gatterbauer, W., and Suciu, D. (2011). Session-based browsing for more effective query reuse. In *Scientific and Statistical Database Management - 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings*, pages 583–585.

[20] Li, H., Chan, C.-Y., and Maier, D. (2015). Query from examples: An iterative, data-driven approach to query construction. *Proc. VLDB Endow.*, 8(13):2158–2169.

[21] Mishra, C. and Koudas, N. (2009). Interactive query refinement. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 862–873, New York, NY, USA. ACM.

[22] Mottin, D., Lissandrini, M., Velegrakis, Y., and Palpanas, T. (2014). Exemplar queries: Give me an example of what you need. *Proc. VLDB Endow.*, 7(5):365–376.

[23] Panev, K. and Michel, S. (2016). Reverse engineering top-k database queries with PALEO. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 113–124.

[24] Psallidas, F., Ding, B., Chakrabarti, K., and Chaudhuri, S. (2015). S4: Top-k spreadsheet-style search for query discovery. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 2001–2016, New York, NY, USA. ACM.

[25] Qian, L., Cafarella, M. J., and Jagadish, H. V. (2012). Sample-driven schema mapping. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 73–84, New York, NY, USA. ACM.

[26] Sarma, A. D., Parameswaran, A. G., Garcia-Molina, H., and Widom, J. (2010). Synthesizing view definitions from data. In *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, pages 89–103.

[27] Shen, Y., Chakrabarti, K., Chaudhuri, S., Ding, B., and Novik, L. (2014). Discovering queries based on example tuples. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 493–504, New York, NY, USA. ACM.

[28] Tran, Q. T., Chan, C.-Y., and Parthasarathy, S. (2009). Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of*

Data, SIGMOD '09, pages 535–548, New York, NY, USA. ACM.

[29] Tran, Q. T., Chan, C.-Y., and Parthasarathy, S. (2014). Query reverse engineering. *The VLDB Journal*, 23(5):721–746.

[30] Zhang, M., Elmeleegy, H., Procopiuc, C. M., and Srivastava, D. (2013). Reverse engineering complex join queries. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 809–820, New York, NY, USA. ACM.

[31] Zloof, M. M. (1975). Query-by-example: The invocation and definition of tables and forms. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 1–24, New York, NY, USA. ACM.