SQL query extensions for imprecise questions

Marie Le Guilly, Jean-Marc Petit, Vasile-Marian Scuturici

SQL query extensions for imprecise questions

Marie Le Guilly*, Jean-Marc Petit, Vasile-Marian Scuturici

Univ Lyon, INSA Lyon, LIRIS (UMR 5205 CNRS), Villeurbanne, France

Abstract

Within the big data tsunami, relational databases and SQL remain inescapable in most cases for accessing data. If SQL is easy-to-use and has proved its robustness over the years, it is not always easy to formulate SQL queries as it is more and more frequent to have databases with hundreds of tables and/or attributes. Identifying the pertinent conditions to select the desired data, or even the relevant attributes, is not trivial, especially when the user only has an imprecise question in mind, and is not sure of how to translate its conditions directly into SQL.To make it easier to write SQL queries when the initial question is imprecise, we propose SQL query extensions: given a query, it suggests several possible additional selection clauses, to complete the Where clause of the query, as a form of SQL query semantic autocompletion. This is helpful for both understanding the initial query's results, and refining the query to reach the desired tuples. The process is iterative, as a query constructed using an extension can also be completed. It is also adaptable, as the number of extensions to compute is flexible. A prototype has been implemented in a SQL editor on top of a database management system, and two types of evaluation are proposed. A first one looks at the scaling of the system with a large number of tuples. Then a user study examines two questions: does the extension tool speed up the writing of SQL queries? And is it easily adopted by users? A thorough experiment was conducted on a group of 70 computer science students divided in two groups (one with the extension tool and the other one without) to answer those questions. In the end, the results showed a faster answering time for students that could

*Corresponding author

Email addresses: marie.le-guilly@insa-lyon.fr (Marie Le Guilly), jean-marc.petit@insa-lyon.fr (Jean-Marc Petit), marian.scuturici@insa-lyon.fr (Vasile-Marian Scuturici)

use the extensions: 32 minutes on average to complete the test for the group with extensions, against 48 minutes for the others.

Keywords: query extensions, imprecise questions, SQL

---

1. Introduction

SQL and relational databases are still widely used to store and access datasets in most commercial data management systems. Indeed, data scientists often use SQL to fetch and explore data. Many companies, as well as many scientific applications, such as chemical research, pharmaceutical applications, and astronomy research, rely on the declarative nature of SQL to explore these massive datasets, to find insights or to select subsets to feed into their models.

SQL views and basic SQL keywords are often enough to write the majority of SQL queries. However, depending on the dataset, several difficulties might arise. The data analyst might not initially know exactly how to formulate her final SQL query, as the conditions to specify might not be directly clear or translatable into SQL. Indeed, the analyst might first think of the necessary conditions in term of adjectives, such as *low, bigger than average, surprisingly high, etc*, that are at first impossible to translate into exact numerical conditions. To do so, the analyst will have to start with a general query, and to try and refine it until reaching the desired output. This iterative process can be tiresome and require many iterations, as the analyst can be overwhelmed with the initial results, and not know what direction to take. Indeed, the initial query might be too general and return many tuples. The analyst has to find a way to understand it, in order to to be able, afterwards, to eventually modify the query. Once a first query is written, the analyst compares the results it returned against her expectations: if they are not reached, the original query needs to be revised. This modification step can be hard, as the analyst has to understand how the given results differ from the expected one. And once the source of the difference is identified, the problem is to find how to modify the initial query to correct it. One solution to reduce the query's output size is to add more selection predicates to the query, to filter some tuples out. Data analysts often adopt a trial and error approach: they try different combinations of attributes and thresholds for a selection predicate, and adapt and modify it according to the result they obtain. The question of queries returning too many tuples has already

been addressed in other research papers, offering various solutions. The interactive query refinement solution exposed in [1] addresses the *too many tuples* problem by transforming the selection predicates of a query with respect to a cardinality objective. The `STOP AFTER` SQL operator proposed in [2] is also a solution to this problem. Finally, *top-k* approaches are also available [3]. In this paper, we propose an SQL-based solution that can be directly integrated in databases management systems, to help users specify their imprecise questions, by suggesting precise selection conditions, to specify the initial query and ultimately identify a relevant result set. The idea is to help the user by summarizing those tuples with several SQL queries, so that she understands better the dataset she is facing. Summarizing data is often done using visualizations, which can be very helpful to understand a dataset. However, they can be hard to do depending on the data type and size, and refining an SQL query based on it can require some work, especially as not all DBMS offer integrated visualization tools. Indeed, going from an image to SQL is not always trivial. This is why we propose to summarize the initial query with several "smaller" (in terms of result size) SQL queries. Moreover, the final objective is to diminish the size of the query's answer set, which requires to add more selection predicates. However, there is a huge number of possibilities for these new predicates, that only increases with the size of the database. Therefore, we propose to summarize the initial query with SQL *query extensions*, that can then be used to drill down the data and deep into it, as they present some possible combinations of additional selection predicates. This way, a user can start by writing a query that contains all the knowledge she has about the data she is looking for: if she does not know anything, she can start with a query returning all attributes from all tables. The extension can then help her to refine her query until she reaches her data of interest. More specifically, given an initial SQL query Q, we propose to a data analyst a list of SQL queries, that addresses two sides of the problem. First, the queries are extension of Q: the beginning of each query is the same, and equal to Q. Then each query has its own set of additional selection predicates, so that each query returns a smaller subset of tuples. Moreover, the extension's results form a partition of Q's results: each extension summarize part of Q, helping the user in understanding what lies into the initial bigger result set.

Example 1.1. *Assume that Alice, a data analyst, has access to the database of a company, which contains several tables, among which the Finance and*

HR tables (with a join attribute EmpID). She is asked to find the gender of employees with a low income: this is typically an imprecise question. Not knowing how to translate the notion of "low" into SQL, and only knowing the existence of a salary attribute in the database, she starts with the following query:

```
Select *
From Finance, HR
Where Finance.EmpID = HR.EmpID
```

She hoped to be able to refine this first query query by looking at the results it returned. However the tuples, presented in table 1 are too many for her to assess what a good threshold would be to get only low salaries. With the solution proposed in this paper, Alice could get help from the three following extensions of her initial query:

```
Select *
From Finance, HR
Where Finance.EmpID = HR.EmpID
  and commission ≥ 6200
```

Extension 1 (4 tuples)

```
Select *
From Finance, HR
Where Finance.EmpID = HR.EmpID
  and commission < 6200
  and sex = 'F'
```

Extension 2 (4 tuples)

```
Select *
From Finance, HR
Where Finance.EmpID = HR.EmpID
  and commission < 6200
  and sex ≠ 'F'
```

Extension 3 (2 tuples)

These extensions contain several pieces of information that can be valuable for Alice. First, they summarize the tuples that were returned by her first query, as they are divided into three queries, and described by the additional selection predicates of these extensions of Q. It gives her directly a selection condition based on the *commission* attribute, which is part of an

EmpID	LastName	Gender	Salary	Commission
e10	SPEN	F	41160	1300
e20	THOMP	M	41250	7400
e30	KWAN	F	39850	5200
e40	SMITH	F	40525	1400
e50	GEYER	M	40175	1100
e60	STERN	M	39560	6200
e70	PULASKI	F	40120	800
e80	FREY	M	40625	6600
e90	HENDER	F	39450	6700
e100	SPEN	M	41560	900

Table 1: Result set of query Q

employee's income. It shows Alice an attributes she had not considered at first, but that is relevant for her question. Moreover she now has a numerical threshold to start from. Finally, the gender selection predicate can also draw her attention to a discrimination she had not considered. Therefore, those extensions are a way to highlight information and patterns that could be pertinent for Alice, and to help her find numerical threshold that can be hard to come up with. This example gives a nice overview of what are query extensions, and what it could be used for. Considering the problem of queries returning too many tuples, we propose the following problem statement:

Problem statement Because the extensions should summarize and refine the considered query at the same time, we propose the following problem statement: given a query Q, and a number of extensions n, return n extensions of Q such that:

1. The results of the extensions are a partition of those of Q.
2. Each extension consists in query Q and one or several additional predicate.
3. Each predicate is on one attribute from the *Select* clause of Q : if the user wants to consider all of them, he can just use a * in the `select` clause, or on the opposite restrict the number of considered attributes by only selecting the ones she is interested in.

Based on such a definition, the problem is then to be able to compute such extensions, so that they can help the user refine her initial query to select the tuples of interest, when the question is imprecise.

It is thus necessary to propose a solution to address this question, and to evaluate how well it is answered. The contributions of this paper are therefore as follows:

- A formal definition of the set of query extensions for a given query
- Given a database, a query and the number of desired extensions, an algorithm to compute such extensions that does not require any laborious user input;
- Visualizations to assist the extension selection;
- An implementation of the algorithm with the design of a web application that can be used by users to test the query extension solution [19];
- Experimentations on the scaling of our algorithms so that extensions can be obtained in a reasonable amount of time.
- User experimentations to validate whether or not extensions are helpful to answer imprecise queries on a relational database.

Paper organization Section 2 introduces the preliminaries and the definitions of SQL query extensions. Section 3 exposes a solution to compute such extensions. Then section 4 presents the implementation, and experimentations that were conducted to evaluate the solution. Finally section 5 summarizes the related work, before concluding in section 6.

2. SQL queries and their extensions

2.1. Preliminaries

Let us start by introducing the basic notations to be used throughout this paper. We assume the reader is familiar with databases notations (see [4] for details). Let $\mathcal{D}$ be a set of constant and $\mathcal{U}$ a set of attributes. We consider a database $d = \{r_1, r_2, ..., r_n\}$ over a database schema $R = \{R_1, R_2, ...R_n\}$, where r_i is a relation over a relation schema R_i and $R_i \subseteq \mathcal{U}$, $i \in 1..n$. We consider SQL without any restriction. We will switch between both languages when clear from context. A query Q is defined on a database schema R and $ans(Q, d)$ is the result of the evaluation of Q against d. In the sequel, to define the extension of any query Q, we will use two operators: π_X the projection

defined as usual with $X \subseteq \mathcal{U}$, and σ_F the selection, where F is a conjunction of atomic formulas of the form $A\theta B$ or $A\theta v$, with $A, B \in \mathcal{U}$, $v \in \mathcal{D}$ and θ a binary operator in operation in the set $\{<, >, \leq, \geq, =, \neq\}$.

2.2. Query extensions

Let's consider an initial query Q, for which query extensions have to be defined. The first characteristics of these extensions is that they aim at refining the initial query Q. Hence, an extension Q_{ext} of Q should diminish the size of the result set. We therefore propose the following definition:

Definition 1. *An extension Q_{ext} of Q is defined by:*

$$Q_{ext} = \sigma_{c_1 \wedge ... \wedge c_n}(Q)$$

where c_i is an atomic formula, for every $i \in 1..n$

The following property therefore follows:

Property 1. *If Q_{ext} is an extension of a query Q then:*

$$ans(Q_{ext}, d) \subseteq ans(Q, d).$$

The second objective of these extensions is to summarize the result of the initial query Q: as one extension will only contain part of the result of Q, it is necessary to propose a set of different extensions, to make sure all of $ans(Q, d)$ is contained in one of the extensions. Therefore, we define the notion of a k-extensions set, containing k extensions of a query Q as a partition of $ans(Q, d)$. More formally, using the definition of a partition:

Definition 2. *A* k-extensions set *of Q over d, denoted by C_Q, is defined as: $C_q = \{Q_1, Q_2, ..., Q_k\}$ such that:*

- *Q_i is an extension of Q, for all $i \in 1..k$*
- *$ans(Q_i, d) \cap ans(Q_j, d) = \emptyset$, , for all $i, j \in 1..k, i \neq j$*
- $\bigcup_{i=1}^{k} ans(Q_i, d) = ans(Q, d)$

It follows that :

- The union of the results of extensions in the set of size k is equal to the set of tuples from the initial query: this way, the initial query is fully represented in the extensions.
- Each extension returns tuples that are not returned by any of the other extensions: this way the options offered to the user are very different from one another, giving us a wide variety of options to choose from.

An example of a 3-extensions set is given in example 1.1. Being able to parameter the number of extensions for the considered query give more freedom to the user, who can explore with different size of extensions sets, and eventually use her domain knowledge to fix the most appropriate size. We will therefore see in the following section how such sets can be computed, in order to provide useful extensions to users.

3. Computation of k-extensions

3.1. General approach

Based on definition 2, it clearly appears that an extension set is a partition of the results of the initial query, where each subset has to be described by a set of selection predicates. Given a query, there exists many different possible partitionings: as a result, it is necessary to decide which one to choose to present to the user, so that it is as useful as possible, taking into account the exploratory context. Given definition 2, there is a first indication of which possible partition to select: as the number of subset k to compute is given in the definition, it therefore removes a vast number of candidate partition. However, it still leaves many possible partitions: the definitions itself is therefore not enough, it is necessary to design a strategy to compute an extension set, that is a coherent as possible with the intended usage of such extensions: in this setting, the solution used to compute the extensions has therefore a huge impact on the result presented to the user, and is a key element of the data exploration strategy proposed to assist the user in query writing.

In addition to deciding which partition of the results to use, it is also important to keep in mind that for each subset option through partitioning, it will be necessary to define the selection predicates returning such a subset of tuples. Given an initial query Q and the number k of extensions, we have to:

- First, divide $ans(Q, d)$ in k disjoint subsets: this is not trivial, and requires to define the division strategy. This first part is related to partioning, as well as to clustering in machine learning [5].

- Then, for each subset, find a query returning all of its tuples, that is an extensions of Q with respect to definition 1. It should be as short as possible to reduce the user's effort, and to help him easily understand the description of the tuples contained in the considered extension. It should also be as informative as possible, explaining how the considered extension is different from the others. This problem is very similar to *query reverse engineering* [6] or *redescription mining* [7].

3.2. Partitioning of the initial set of tuples

3.2.1. Using clustering

Let's first consider the partitioning of the initial set of tuples $ans(Q, d)$: the partitions should have a sense for the user, and return tuples that are meaningful with respect to the exploration process. As a result, we propose to identify tuples that make sense together, and that identify regions of $ans(Q, d)$ that share some common characteristics. These regions are more likely to be of interest because they are more likely to correspond to the need of the user, who is usually not looking for sets or random tuples. In addition, these common characteristics are likely to help the formulation of the selection predicates for the extensions, as they can rely on these common characteristics of the tuples.

As a consequence, we propose to divide $ans(Q, d)$ by grouping together *similar* tuples. More specifically, we propose to divide the initial set of tuples using a clustering algorithm (see [8] for an overview), as it corresponds to all of our requirements: The clustering algorithm will group *close* or *similar* tuples together; Clusters are pairwise distinct, so the sets will not overlap; and the clusters cover $ans(Q, d)$. Regardless of the initial query Q, the answer set of the query will always be a unique table, that can then directly be sent to the clustering algorithm, along with k, which directly corresponds to the number of clusters to be produced.

3.2.2. Challenges of clustering

Clustering is one solution to address the considered problems, but it yields intrinsic challenges that we ought to address. The main issues is that because we propose a general approach that should be applied to many different

databases and queries, the data to cluster is initially unknown: it is therefore necessary to take some precautions. First, it is important to normalize the data to cluster, to avoid the features with higher values to take more importance than the other. To this end, the data is standardized feature-wise, meaning it is centered around the mean and scaled to the variance.

We also have to make the assumption that the number of features stays reasonable: indeed, clustering can then be less significant, due to the curse of dimensionality [9]. Finally, most clustering algorithms require to be able to define the distance between two tuples, which can be more or less easy depending on the feature's types. From all of this, it is clear that clustering has to be manipulated carefully: in our setting, it however provides an efficient solution to group together similar tuples, that are more likely to have interest for the user.

It is also necessary to choose the specific clustering algorithm. Given the input dataset, a clustering algorithm groups together tuples that are *close* to one another, which requires to have some sort of distance between tuples. If this is straightforward for numerical values, with for example the well-known euclidean distance, a database is likely to contain different datatypes: therefore our solutions requires a clustering algorithm that is able to handle mixed datatypes.

Additionally, one key element in choosing the clustering algorithm is that the number of clusters to be produced in known: in our extension setting, we consider this parameter to be given by the user with respect to her exploration requirements: this therefore allows her to explore different extensions possibilities with different values for this parameter.

Considering all these parameters, we propose to compute the k sets of tuples using the *k-means* clustering algorithm [10], that can take the number of clusters to produce as an input: this solution was also used in [11] to propose visual data exploration of query results. Moreover, to handle both numerical and categorical attributes, we propose to use its variant *k-modes* [12].

Moreover, as the user might not always know the size of the extensions set she desires, we offer to automatically try different ones: she can therefore specify a lower and upper bound (by default for k = 2 to k = 10), so that different values of k can be tested, and therefore several extensions sets proposed. As among all the computed ones, some extensions sets might be more pertinent than other, they are ranked according the clustering quality, based on the well-known clustering score of the silhouette coefficient [13]:

let's consider a tuple i assigned to cluster C_i. Then we define:

- $a(i) = \frac{1}{|C_i|-1} \sum_{j \in C_i, i \neq j} d(i,j)$
- $b(i) = \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i,j)$

The silhouette score for i is then defined as:

$$s(i) = \frac{b(i)-a(i)}{\max\{a(i),b(i)\}}$$

Considering all the tuples that have been clustered, the silhouette coefficient is defined as:

$$S_{sil} = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{|C_k|} \sum_{i \in C_k} s(i)$$

Where $d(i,j)$ represents the distance between two tuples i and j.

Intuitively, it is the average of how close the points in the same cluster are, and how far they are from he others cluster: the better the clustering, the more separated the clusters.

All these elements allow us to obtain clusters, even though it should be kept in mind that for some datasets, the clustering tendency can be limited. However, our main objective is not to provide a clustering tool, but to give suggestions to a user who does not know how to refine her query : as a result, even though the data returned by an extension might be scattered and not so clustered, it is still a possible indication of how to refine at this point of the process. Moreover, because we use kmean as a clustering algorithm, we will obtained the imposed number of clusters, even though the cluster tendancy. Finally, the silhouette coefficient is therefore also a way to assess the quality of clustering, allowing the user to modify some parameters if it does not appear satisfying to her.

3.2.3. Preparation for extensions construction

After the clustering, each tuple can be assigned to a single cluster, that can be added as an additional attribute to $ans(Q,d)$. This cluster column acts as a form of tuple labeling, indicating which tuples are likely to form an interesting subset, because they share some common characteristics. Compared to other existing labeling techniques, this clustering-based approach has several advantages. First, it drastically simplifies the role of the user,

EmpID	LastName	Sex	Salary	Commission	*Cluster*
e10	SPEN	F	41160	1300	*2*
e20	THOMP	M	41250	7400	*1*
e30	KWAN	F	39850	5200	*2*
e40	SMITH	F	40525	1400	*2*
e50	GEYER	M	40175	1100	*3*
e60	STERN	M	39560	6200	*1*
e70	PULASKI	F	40120	800	*2*
e80	FREY	M	40625	6600	*1*
e90	HENDER	F	39450	6700	*1*
e100	SPEN	M	41560	900	*3*

Table 2: Tuples from table 1 labelled by clustering ($k = 3$)

who does not have to endure the boring task of manual labeling. Additionally, this solution goes beyong the simple binary labeling that is usually used: instead of just considering a tuple as interesting or not, it says whether or not it is interesting to consider it with other tuples or not. This way, instead of evaluating individually the relevance of a tuple, this allows to consider groups of tuples that might be relevant for a given task, together. In practice, a new attribute, called `cluster`, is added to the schema of the query to keep track of the cluster corresponding to each tuple.

Example 3.1. *Table 2 presents tuples from table 1, with the additional attribute* cluster *that is the cluster tuples have been assigned to, for a 3-extensions set.*

3.3. Construction of extensions for each subset of tuples

3.3.1. Using a binary decision tree

The second part of the process aims at finding, for each cluster, a set of selection predicates that can describe it. Once again, in an exploration setting, there are some specific considerations to take into account: there is a trade-off to find between the accuracy of the clusters description, and the *utility* of the extensions. Indeed, they should be concise enough, so that the user can understand the data they describe quickly, and as informative as possible, in order to underline what separated a specific cluster from the others.

Taking this constraints into consideration, we propose to use a decision tree to construct the query extensions, using the tuples as a classification dataset, and the cluster column as the class to predict. The general idea is to learn what distinguishes a cluster from another: the decision tree will identify relevant attributes and discriminating values to describe concisely a cluster with respect to the other ones. Each leaf of the tree can then be considered as an extension, by using the selection conditions that lead to it from the root of the tree.

Using decision trees to generate SQL queries is a technique that has already been exploited [14]. To be able to reach the objectives defined for our SQL extensions, we follow the same path but with *binary decision trees* (BDT) [15], which is a tree splitting at each node on exactly two opposite conditions.

One specificity of our approach is the fixed number of extensions to be computed: we need exactly k extensions. For the clustering, this meant computing k clusters: however, the decision tree might produce more leaves than classes. If we want to produce exactly k extensions, two solutions are considered, wand we let the user decide which strategy she wants to use:

- As they are k classes, it is possible, for each class, to consider all the leaves of the tree that corresponds to it: the different conjunction of selection conditions leading to each leaf can then be disjoined, to give only one selection predicate for the considered class. In this case, the produced extensions might be longer and less easy to understand directly because of disjunctions and conjunctions, but maybe also more informative.

- The other possibility is to limit the growth of the tree, to only allow it to have k leaves: by considering each leaf as an extension, k extensions are therefore obtained. In this case the extensions might be shorter and easier to understand, but also less discriminating. This second solution requires to be able to produce a decision tree with a constrained number of leaves, meaning we have to adapt the decision tree algorithm so that it is constructed taking the constraint into account. We address this problem in the next section.

It should be noted that the choice between the two strategies is up to the user, and has an important influence on the decision tree. If it is allowed

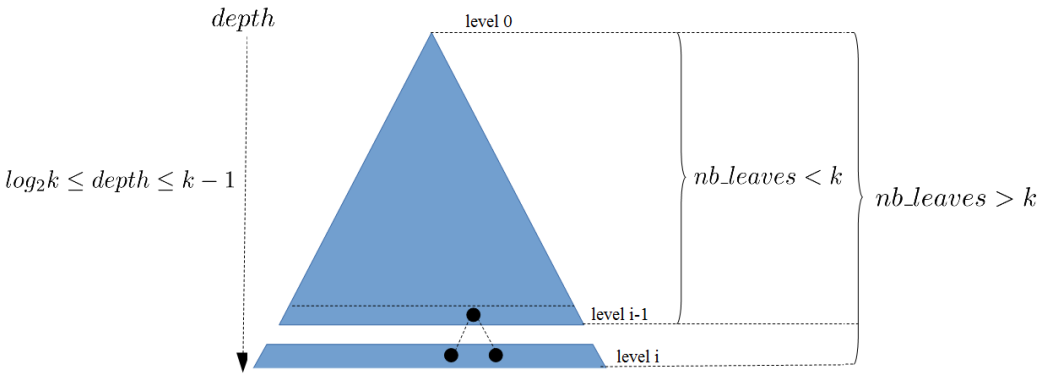


Figure 1: Construction of a binary decision tree given a fixed number of leaves

to fully grow, it is more likely to overfit the data, but also to produce more detailed extensions with more selection clauses that can be valuable. On the opposite, the tree with a constrained number of leaves is less prone to overfitting the data, and will have more concise, but maybe less informative selection predicates. It is therefore all about balance, and it is important to underline that the tree is here only a way to go from the partitioning of clustering to SQL selection predicates: the best strategy is therefore the one most suited to the user's need, with respect to the understandability and utility of extensions for the considered SQL task.

3.3.2. Obtaining a constrained BDT with k leaves from a given data partition

Constrained generation of BDT given a specific number of leaves has been studied in [16]. In our case, we just need to explore levelwise the search space (breadth-first search) and stop as soon as the number of leaves exceeds k. To do so, we rely on the following property of BDT: if N is the number of classes to classify, the depth of the BDT is bounded between $\lceil \log_2(N) \rceil$ and $N-1$. Both bounds are attainable: the first one with a *full binary tree* and the second one with a *right deep tree*. For the query extensions, it requires to stop somewhere in-between.

To reach exactly the k leaves constraint, the construction of the tree starts as usual, level by level. At each new level, there are then three possible scenarios:

- The total number of leaves in the tree is inferior to k: then the construction of the tree can continue.

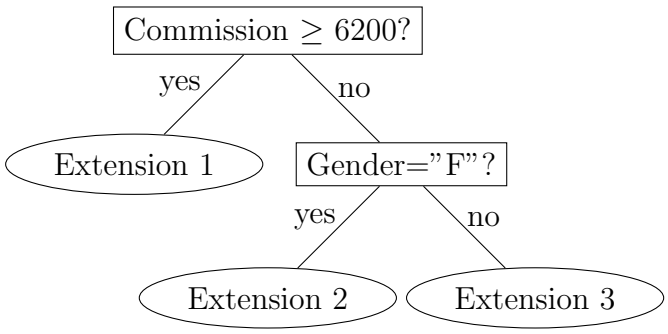


Figure 2: Binary decision tree from Table 1

- The total number of leaves in the tree is equal to k: then it is perfect, the process should stop there.
- The total number of leaves in the tree is greater than k: it means the last level has added to many leaves, and some should be removed.

To figure out how to deal with the last scenario, and how to remove leaves from the tree, let's consider figure 1: assume the number of leaves in the BDT is less than k at level $i-1$, and greater than k at level i. The process to remove the unnecessary leaves works as follows: while the number of leaves remains greater than k, replace two leaves at level i from the same parent by turning this parent into a leaf at level $i-1$ (using a majority vote to assign a class to this new leaf). As two leaves are replaced by one, this process removes the leaves one by one, until the total number of leaves in the tree is equal to k. To chose which node to remove, we use the impurity of the node, in order to first remove the nodes that are not discriminant (high impurity).

Example 3.2. *From the clustering in table 2, the binary decision tree of figure 2 can be obtained. In this running example, the decision tree leaves match exactly with the clusters.*

The example points out that the clustering and the binary decision tree may coincide. However, this is not always true since some tuples may fall into the wrong cluster or some clusters could be lost by the binary decision tree because, as previously explained, we do not grow a full tree and limit the number of leaves. It is possible to obtain a correspondence between clustering and decision tree by using our alternative strategy, which allows to make disjunctions and therefore to fully grow the tree.

3.3.3. Obtaining SQL statements from a BDT with k leaves

Once the binary tree has been constructed, each leaf can be reached through a unique decision path. The decision path from the root of the tree to a specific leaf can be written as the conjunction of each decision encountered along the road. It is therefore straightforward to go from a decision tree to a SQL query. After exploring all the path in the tree, each conjunction can be directly injected in the *where* clause of a SQL query, and gives a new extension.

Example 3.3. *From the decision tree on figure 2, every extension from example 1.1 can be obtained easily.*

3.4. Algorithm proposal

In order to combine all the steps described previously, and to specify how the k-extensions set of a given query is to be computed, algorithm 1 is proposed hereafter. It takes a query Q and a database d as an input. In addition, it also takes a lower and upper bound for the size of the extensions, and allows to select the strategy for the decision tree: by default (when $tree_{strategy} = false$), the extensions only contain conjunctions, and the number of leaves in the tree is therefore limited. All the parameters for the algorithm allow for a good adaptability to the user's need, in order to be able to select the most appropriate extension.

The algorithm works as follows:

- The algorithm is centered around function `Extend`, that computes an extension set for a given query on a database, given the number of extensions to compute and the selected strategy for the tree:
  - Line 12, we compute the result of the initial query. Clearly, if the size of the result is expected to be large, the online computation of extensions might take some time, and sampling could speed it up. Details on scaling and sampling are discussed in section 4.
  - Line 13, the clustering transforms the dataset wd into a labelled dataset lwd in which each tuple is labelled with the cluster it was assigned to. The quality of the clustering is evaluated line 14.
  - The decision tree is then built from lines 15 to 21, depending on the selected strategy. The conditions are extracted from the tree.

  - The conditions are sorted by length. They are then turned into extensions, and stored.
  - Finally, the considered $k - extensions$ set is returned along with the corresponding clustering quality.

- The main algorithm (lines 2-10) computes the extensions sets for the different values of k. The different extensions sets are ordered according to the quality of their clustering.

Thanks to the parameters of the algorithm, the user can have more influence on the extensions. However, the default parameters are already a good start for a novice user. This flexibility makes the extensions useful for different user profiles and different kind of data selection tasks.

The conditions from definition 2 are satisfied by the proposed algorithm as stated in the following property.

Property 2. *Let d a database over R, Q a query over R, and k an integer. $extension(Q, d, k)$ is a k-extensions set of Q, i.e for $\{Q_1, Q_2, ...Q_k\}$ in $Extend(Q, d, k)$, and for all $i, j \in 1..k$, $i \neq j$:*

$$Q_i \text{ is an extension of } Q \tag{1}$$

$$ans(Q_i, d) \cap ans(Q_j, d) = \emptyset \tag{2}$$

$$\bigcup_{i=1}^{k} ans(Q_i, d) = ans(Q, d) \tag{3}$$

Proof. (1) By construction, and with respect to definition 1, Q_i is an extension of Q

(2) and (3) By definition, a decision tree builds a partition. Properties 2 and 3 directly follow from the definition of a partition.

□

4. Implementation and Experimentations

4.1. Algorithm implementation

Algorithm 1 was implemented in Python 3. For sake of simplification, the implementation is for now limited to numerical attributes. The *kmean* algorithm is taken from the *scikit-learn* [17] library, as well as the decision tree based on the CART algorithm [18].

Algorithm 1: Query extensions sets configuration procedure

```
1  procedure Extension (Q, d, k_min = 2, k_max = 10, tree_strategy = false);
   Input  : A query Q over R,
            d a database over R,
            k_min the lower bound of query extensions set,
            k_max the upper bound of query extensions set,
            tree_strategy the strategy for the decision tree (true if it is
   different from default one)
   Output: ext_list a list of k extensions sets of Q, with sizes from k_min
            to k_max
2  ext_list = [] // the final list of extensions sets
3  scores = [] // list of clustering quality for each extensions
     set
4  for i = k_min; i ≤ k_max; i++ do
5  |  S_c, quality = Extend(Q, d, i, tree_strategy)
6  |  ext_list.append(S_c)
7  |  scores.append(quality)
8  end
9  ext_list = sort(ext_list, scores); // sort the extensions sets by
     silhouette score
10 return ext_list;
11 Function Extend(Q, d, k, tree_strategy = false):
12 |  wd = ans(Q, d) // wd:  working data
13 |  lwd = kmeans(wd, tree_depth) // lwd:  labelled wd
14 |  quality = silhouette(wd, tree_depth) // silhouette:
   |   clustering quality
15 |  if tree_strategy then // allow any depth for the tree
16 |  |  tree = DecisionTree(lwd)
17 |  |  conditions = getDisjunctionConjunction(tree) // get one
   |  |   extension per cluster
18 |  else
19 |  |  tree = DecisionTree(lwd, k)
20 |  |  conditions = getConjunction(tree) // get one extension
   |  |   per leaf
21 |  end
22 |  S_c = {}
23 |  conditions = sort(conditions); // (sort by length)
24 |  foreach c in conditions do
25 |  |  S_c = S_c ∪ σ_c(Q)
26 |  end
27 |  return S_c, quality;
```

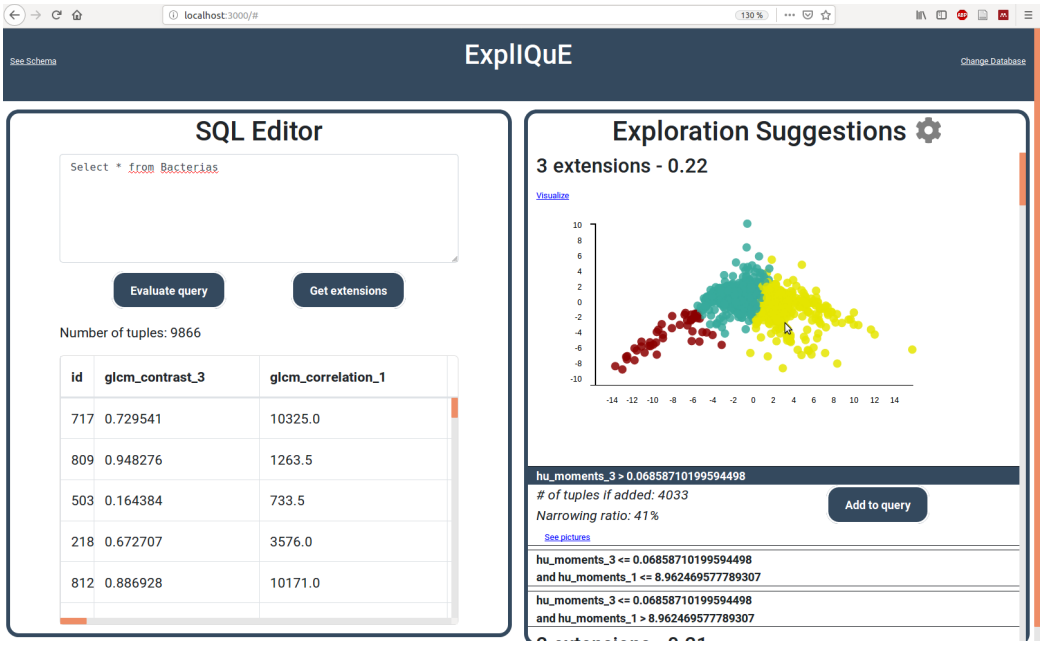


Figure 3: SQL query extension prototype

4.2. SQL Editor Prototype

We also implemented a web interface, presented on figure 3, that was presented as a demonstration in [19] (demonstration available online[1]. It allows to connect to any available database stored with MySQL or Oracle, and provides all the basic functionalities of a SQL editor: seeing the schema, querying the database, browsing the results. But it also provides extensions-related functionalities, the first one being proposing a set of extensions for a user defined input query. In addition, visualizations are available to assist the user in choosing an interesting extension. The first is a scatterplot of the results of the query being extended, with tuples are grouped by extension. The data is projected on two dimensions using PCA, and each extension is presented using a different color. Moreover, the visualization is interactive, as the user can see the extension corresponding to the datapoints by moving the mouse over the scatterplot. The purpose of this visualization is to show in one glance to the user the size and dispersion of an extension, as well as how separated each extension is with respect to the others. In the

[1] `https://youtu.be/oK8xWGCWj_A`

specific case of image databases, where tuples are associated with images, another visualization is possible. The images associated to the results of an extension can be displayed as a mosaic in ExplIQuE. Finally, extensions can be configured: number of extensions to compute, methods for decision tree exploration. The interface allows to connect to any available database, and provides all the basic functionalities of a SQL editor: seeing the schema, querying the database, browsing the results.

4.3. Experimentations

Two different types of experimentations were conducted. First, the scaling of the algorithm was studied, to study the response time based on the database's size. Then a user study looked at how users benefited from the extensions when exploring a database, and how well they adapted to the tool.

4.3.1. Scaling experimentations

As the extensions are computed online, one crucial point of the system is to be able to return the extensions in a reasonable amount of time. As mentioned in section 3, the solution to limit the response time of the algorithm is to sample the tuples before computing the extensions: this allows to obtain a trade-off between user waiting time and the extension's precision. The experiments were run using a machine with an Intel Core i7-7600U (2.8 GHz) CPU and 16GB of memory. To determine what parameters influence the response time, and the appropriate values for the scaling, we designed an experimentation to explore the influence of the parameters on the extension's computing time. The scaling experimentations were conducted on a database with data from the Large Synoptic Survey Telescope (`https://www.lsst.org/`) containing 500 000 tuples over 25 attributes. It only contains one table, as the number of tables only influences the evaluation of the initial query, but not the extension process *per se*. The influence of three parameters was studied, with respect to the extension's computation time: number of tuples, number of attributes, and number of extensions to compute on figure. These parameters were studied two by two, to study every aspect of their influence, and are presented on figure 4a, 4b and 4c.

From figure 4a, it follows that the number of attributes barely influences the computation time. The number of extensions does seem to slightly increase it, which is also visible on figure 4b, but it is negligible compared to the influence of the number of tuples: on figure 4c, it is obvious that it is the most influential parameter. It is also clear that for a high number of tuples,

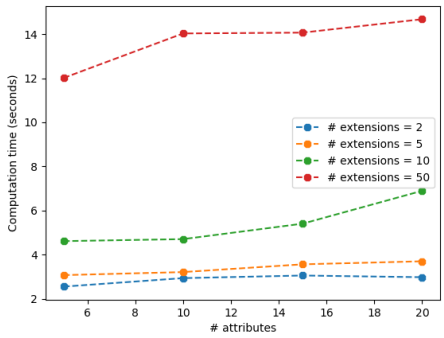
the time necessary to produce the extensions is not acceptable for an online system, as users will not accept to wait more than a few seconds to obtain their results. This confirms the necessity of sampling, in order to provide the extensions in a reasonable amount of time.

Using these results, it appears that up to 50 000 tuples, the extensions are computed in less than 20 seconds. But to determine a good sampling ratio, the quality of the extensions should also be taken into account: even if they are computed quickly, they should still be a good representation of the extensions obtained when taking all the tuples into account: to this end, a second round of experimentation was conducted, to evaluate the impact of sampling over the extension's results. First, the extensions for the query returning the entire database (500 000 tuples) were computed.

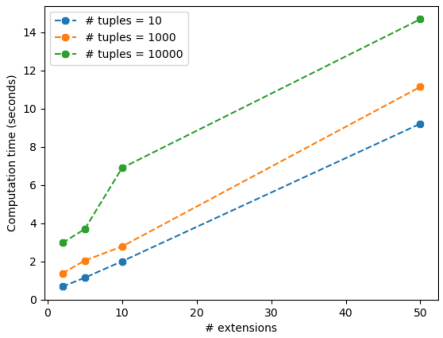
Then, for the same number of tuples, the extensions were computed, but over a sample of the initial query's result, using different sampling sizes (100, 1000, 5000, 10 000, 25 000, 50 000). Finally, the obtained extensions were compared to the ones obtained without sampling, by comparing how they partition the initial tuples, using the adjusted rand index [20]. In addition, two sampling strategies were compared: random sampling, and systematic sampling (see [21]). The results of these experimentations are presented on figure 4d. Systematic sampling requires slightly more tuples to reach the same behavior as random sampling. However, both samplings rapidly reach good results in comparison to the full data extensions, as with a sampling of 1000 tuples, the adjusted rand index is already above 0.8 (1 meaning results identical to non sampled dataset). With 10 000 tuples, the score reaches 0.98 with random sampling, and the computation time with respect to figure 4c is around 5 seconds. These results show that sampling can considerably speed up the extension computation while keeping a very acceptable quality. Those results were confirmed on several other databases not presented in this paper, showing that sampling provides an excellent trade-off between computation time and extension's quality.

It should also be noted that in order to speed up more the extensions computation, it is possible to store the results obtained for one query, in order to reuse it later to avoid recomputing extensions that had already been obtained before. As a result, for each query, it is possible to look in the query log to make sure its extensions are not already saved in the system. Finally, as the most expensive part of the process is the clustering, that requires to compute distances between each pair of tuples, we also propose to save the distance matrices, so that they can be directly reused in other extensions

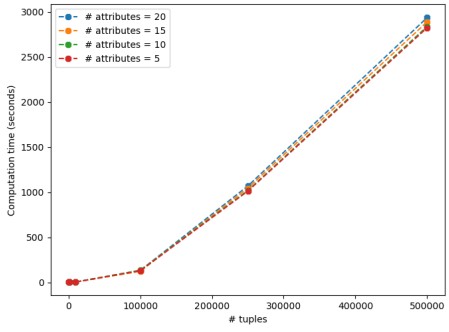
computations. Using these two methods, even when an extensions set takes some time to compute, it will only happen once, and it will also speed up extensions computation for following similar queries.



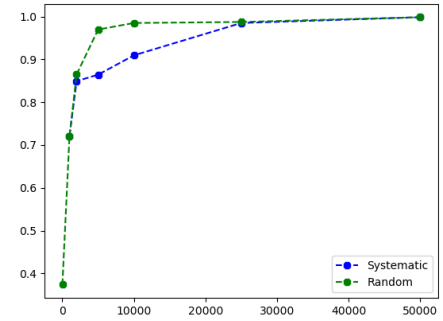
(a) Extensions computation time vs number of attributes and extensions (10 000 tuples)



(b) Extensions computation time vs number of extensions and tuples (20 attributes)



(c) Extensions computation time vs number of tuples and attributes (10 extensions)



(d) Comparison of extensions quality against the sampling size

Figure 4: Experimental results for the scaling of extensions computation

4.3.2. User experimentations

The objectives of this experimentation was to see the benefits such a tool could bring if integrated in a DBMS. It was decided to explore two different categories of measures : In terms of writing time, is it faster to reach a desired

set of tuples using extension ? And how well is the extension tool accepted by users ?

Organization. An SQL competition was organized, with a group of 70 computer science students (last year bachelor students and master students), using a version of the interface anterior to the one presented in figure 3: it did not include the visualizations. All students had at least basic knowledge in SQL and data management. They were initially only told that they would have to address several SQL-related challenges.

Prior to the experiment, participants were randomly divided into two groups. The division was however balanced in terms of number of students from each level (bachelor, first year and second year master students). The experiment required to evaluate SQL queries on a database. For this, the first group (referred to as group EXT from now on) had access to the extension tool, while the other (referred to as group NoEXT from now on) had a tool that was designed to be similar to the one of group EXT, but without the extension possibility. This disposition was chosen to be able to compare the results of the two groups, i.e to see the difference between groups with and without extension, while working under similar conditions (softwares with similar functionalities in terms of classic querying tools). Each group was asked the same ten questions on a database (see appendix)

Design of the test. To test the extensions interface, we had to design *imprecise* questions, to put the students in the situation for which the extensions were designed. The design of these questions was delicate, to find the right level of difficulty. When conceiving the questions, our purpose was to propose a fair situation for groups EXT and NoEXT. For this reason, we eliminated several types of questions : questions that were trivial with extension, but very difficult to do without it, and questions for which extension has no interest: queries with empty result sets, dates comparison, specific operators from DBMS... All questions exposed a scenario, and then asked to find out the SQL query to solve it. The questions were separated into two categories . The first three were classic SQL queries, that are directly and easily transformable into SQL queries, on which the extension tool was not useful. They were used to verify that each participant really had basic SQL skills, and that groups EXT and NoEXT had similar results, and were therefore well balanced. The other questions (number 4 to 10) were the imprecise ones: their conversion into an SQL query was not straightforward. Their specification was less strict, as selection conditions were not specified in terms of

numbers, but using adjectives such as *higher, bigger, lower, above average, low, etc.*

Test setup. To guide participants, we indicated the approximate number of tuples the query was supposed to return. For some questions, we proposed data visualizations and asked participants to formulate queries that would return a part of these visualizations. The objective was to transform a visual pattern into a query, so they had to identify the pertinent conditions to characterize the given pattern. This part of the experimentation then inspired the additional functionalities for the interface. Participants had one hour to answer the 10 questions. They use the tool to write queries and evaluate them on the database, and once they thought they had the right query, had to submit it online. They were not told whether their answer was right or not, as in real-life where only the data analyst can know if she obtained the data she wanted. During the hour of experimentation, we were able to monitor the time each participant spent on each question. After the experiment, we also checked whether the answers they submitted were correct or not. Moreover, we were able to say, for each question, if participants from group EXT had used extension or not. At the same time, group EXT had to adapt to the extension tool, and they did not receive any specific training on how to use the tool before the test. They did not get any additional time, and had to use the hour to both answer the questions and master the extensions. This was done to avoid influencing them on their use of the extension tool, and to see how they would adapt to this new functionality.

Threats to validity. There exists several threats to the validity of our experiments. The main threats are related to the selection of the participants and their distribution into two groups. Indeed, if the distribution is not random, especially in case regarding their SQL's knowledge level, the obtained results could be affected by the students characteristics. To prevent this, we randomly assigned students from each level (bachelor or master) into the two groups. Additionally, another related threat is the number of participants: we had to make sure to enroll enough students in order to obtain representative results. Moreover, there might be a bias due to the fact that participants knew that they were in a experiment, which might influence their behavior: for example, they might change how they look for an answer to exploit the experiment's setting. to limit this threat, we reproduced conditions similar to the examinations students are used to, and fixed a limited answering time

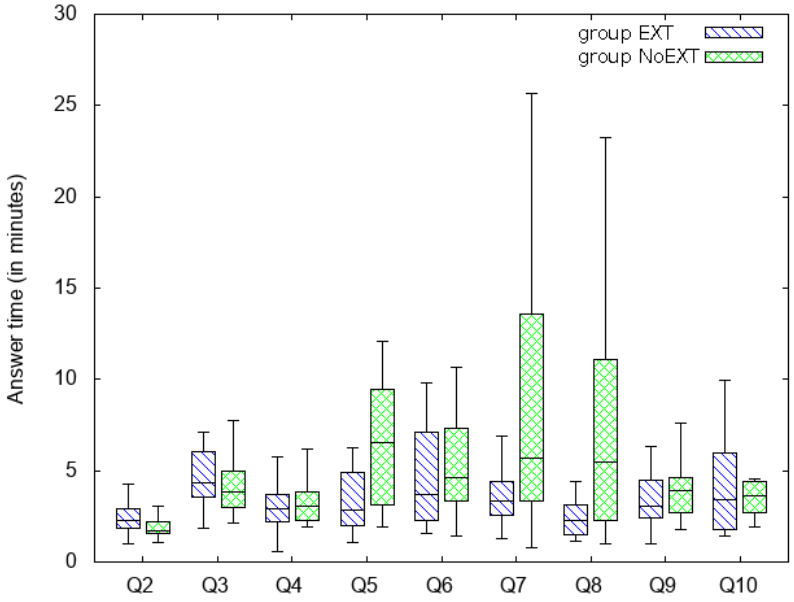


Figure 5: Boxplot of answering time per question, for groups EXT and NoEXT, only for correct answers

so that they would use the most efficient way for them to answer questions. Finally, it can argued that one limit of our experiment is the representativeness of our test sample, as our experiments was done with students. Because of their various SQL levels, we had diversity in our sample, with students with various backgrounds, and representative of our target for query extensions. But additional experiments with other diverse participants could be interesting to strengthen the representativity of the presented results.

Results. To analyze the impact of query extension on the time necessary to answer the questions, the first result that is interesting to look at is how much time each group spent on average answering each question: those results are presented on figure 5. Only correct answers were taken into account on this figure, as participants who provided a wrong answer might have spent a lot of time on a question looking for the answer without finding it, or on the contrary given up quickly if they did not know how to find the answer. There are several interesting points to notice figure 5. For questions 1 to 3, the results of the two groups are similar, which was the initial objective. When extension was not necessary, the performance of both groups were

equivalent. Question 4 was still easy for both group, as could be expected as the visualization was here to help . Even though the use of extensions could have helped on this question, it does not seem to have made a difference, as the average answering time is very similar for both groups. This means that a good and efficient visualization, when efficient, can also be useful. For questions 5 to 10, the difference between the two groups is much more important and it is clear that group EXT performed considerably faster than group NoEXT. This is a strong argument to support the fact that SQL query extension can indeed make the SQL query writing faster. The difference is stronger for questions 7 and 8, which seem to have been the most difficult questions for participants. Finally, the results of group EXT are much more packed than for group NoEXT: participants who had access to extension had a way to help them if they were stuck on a question, contrary to group NoEXT participants who had to search by themselves until they identified the answer. This is flagrant once again for question 7, where someone spent more than 25 minutes looking for the answer. To summarize, when evaluated in similar conditions, the group with access to extension performed faster than the group with only classic SQL tools.

In the future, it would be interesting to develop other experiments, for example on real databases, by enrolling domain experts that have a good understanding of the data but might need help when confronted to imprecise queries.

As mentioned previously, it was also possible to say whether a participant had used extension for a given question or not. In total, 70% of participants used query extension at least once, while the others completed the test without using it. We analyzed the way participants had used the extensions: on figure 6, interesting patterns can be observed. The main observation to do is that once participants have used query extension for a question, they are very likely to use it again in the next question. This is indicated by the continuous blue lines on this figure. This is a really important result, as it showed that once a user has understood the utility of extension, she will use it again. This observation is particularly true for participants number 1 to 13, which in addition did not make many mistakes. Participants 14 to 19 also used extensions a lot after their first use, but made more mistakes: when looking at their answering time, it seems that they did not have much time to complete the last questions, and might have been in a rush and did not give correct answers. Finally, participants 20 to 24 seem to have tested extension, but preferred to finish the test without using it.

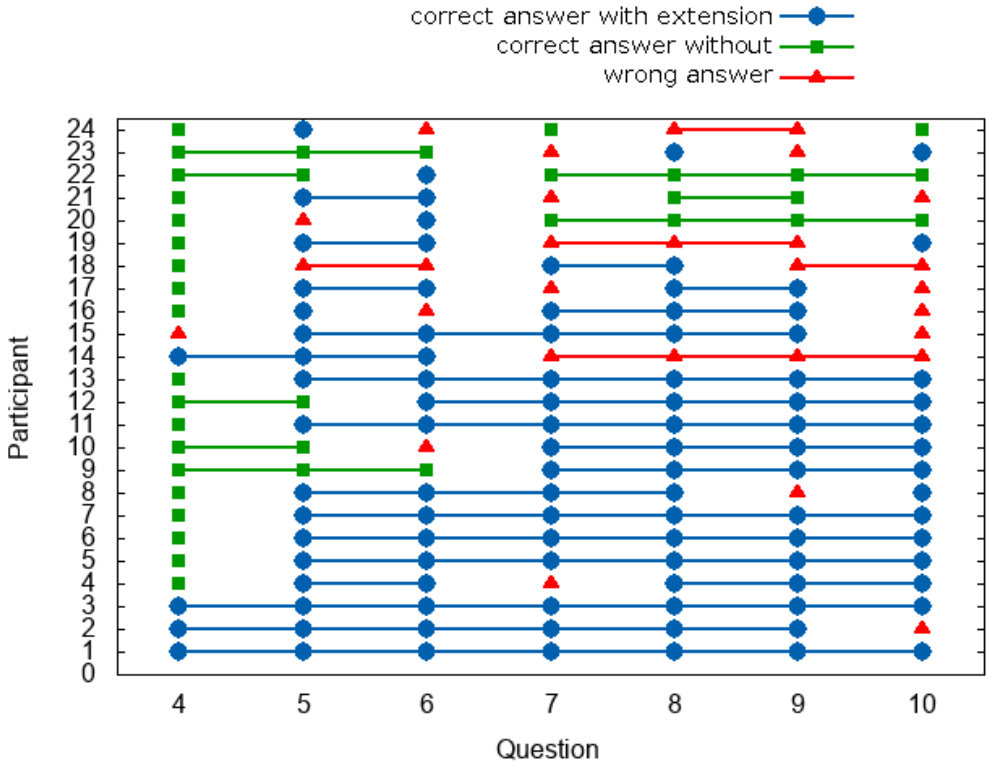


Figure 6: Type of answer per question, for participants who used query extension during the test

To summarize, we demonstrated that the group with the extension tool answered imprecise questions faster than the one without: on average, group EXT completed the test in 32 minutes, against 48 minutes for group NoEXT. This is not only because the tool allows to write faster, but mostly because it identifies conditions that take much more time to find manually, and helped students that were confronted to queries returning many tuples. We saw that the tool is well accepted, and showed that the use of extensions was not a single isolated try by participants, but that a first use encouraged them to use it again.

5. Related work

As far as we know, SQL query extension is a new problem that has not been studied yet. Nevertheless, related contributions exists such as [22],

that proposes a context aware autocompletion tool for SQL. Even though it suggests completions in various SQL clauses, the completions offered rely mostly on the schema and on the database's log, and does not look at the data itself contrary to our approach that relies on the database's content. Also related to the notion of imprecise queries is *vague* queries, introduced in [23]: these are queries in natural language, that can be both vague in terms of attributes to selection, or projections conditions. The author propose a solution to answer such queries, using metrics to also retrieve closely related results. In comparison, they do not allow a direct SQL translation, and expend the possible results instead of refining them as exposed in this paper.

In addition several active research fields are closely related to this paper.

Query inference and example tuples Many approaches try to infer query based on *example tuples*, manually labelled by the user, usually as positive (if it should appear in the result set of the query) or negative. Among those we can cite [24] that suggest a set of queries returning such example tuples, or [25], where the objective is to infer the join query returning the expected result. Those approach are similar to ours in the sense that their purpose is to help query formulation. However, the labeling done by the user is a additional task she has to do in addition to the usual ones. In comparison, we only ask an input query from the user, which is something she has to do anyway in order to get results in the database. The clustering phase of our solution is in charge of the labeling, which means many more tuples can be labeled as it is automatic. Other approaches also exist to avoid manual tuple labeling, such as [26], that learns queries to help users in data exploration, using genetic programming algorithms. Some approaches from natural language processing are another alternative to interact with databases, such as in [27], or like [28] that highlights part of a natural language query to explain the non answers in the translated SQL query.

Interactive data exploration Many recent works concern interactive data exploration in database, with techniques helping users understand and discover their data using machine learning. Some examples of such works are exposed in [29]. Many of those approaches rely on manually labelled tuples : we can cite the *AIDE* framework offers [30] that learns what tuples are of relevance for the user and which are not. The machine learning phase is essentially based on decision trees and SVM. Thereafter, this process was improved in [31], by using even more machine learning. This second paper uses the same framework, but completes it by identifying underlying user habits based on their labeling. Those habits are turned into attributes used in a

clustering. This way, similar users are identified, which is used for speeding the process by using previous data exploration by similar users to give even more relevant tuples. We can see here many similar features with our query extension proposal. However once again those approaches require more work from the user.

Faceted search The goal of extensions is to guide the user to various parts of her initial data, by describing it using a set of various attributes, that highlight different aspects of the dataset. In that sense, this is related to faceted search and exploration [32], that allow a user to explore and narrow her results with facets conditions that work as restrictions on attribute values. With extensions, we propose different possible restrictions with several combinations of attributes and values. *FleXplorer* [33] proposes a framework for faceted search, that can be used for relational databases. An other solution, *YMALDB* [34], offers to explore databases by recomanding "you may also like" results, that where not part of the user's initial query. Theses additional items are computed by identifying interesting sets of attributes and values (facets).

Redescription mining One part of our proposed solutions aims at describing a set of tuples,mainly using a decision tree. This related to *redescription mining* [7], that unifies considerations of conceptual clustering, constructive induction, and logical formula discovery. Nevertheless, they do not consider at all SQL queries as we do, and are interested in enumerating all possible redescriptions verifying some conditions, with enumeration techniques quite different from our proposition. Decision trees in databases have also been studied in various forms: in [14] they are used to reformulate in query for data exploration. We can also mention works on integrating decision trees into databases as objects that can be stored and queried, such as in [35] or [36]. Also related to our solution are predictive cluster trees that combine those two method into one [37].

Reverse query engineering Finding a query returning a given set of tuple is also closely related to *reverse query engineering* [6, 38] that considers the following problem: given a tuple set T in a database d, find a query Q such that $ans(Q, d) = T$. Many theoretical results exist with respect to the language permitted to express Q, conjunctive queries and variants. This is what we do with the decision tree used to formulate a query returning the tuples from a cluster. However, in our context, some simplification is permitted since part of the query is known.

Machine learning and relational databases There is a part of research

trying to bridge the gap between machine learning and database. [39] argues that bringing databases and machine learning algorithms closer might only be beneficial in terms of performance. More concrete applications of this has actually been done, such as in [40] where an entire machine learning library has been adapted so that it is compatible with a storage of data in a DBMS instead of a data structure in main memory. Moreover, they also adapted the algorithms in order to make use of native SQL operators. There is also [41] which is a SQL extension for data mining.

6. Conclusion

In this paper, we proposed to address the problem of extending SQL queries to answer imprecise questions. Without any intervention required for the user, the query can be extended automatically with the number of extensions asked by the data analyst. The extensions summarize the initial dataset, and helps the user to understand how her data is organized. Moreover, they can be used to refine the query and reach regions of interest. The approach is based on classical machine learning algorithms, adapted to fit into the definition of the extensions we have proposed. A SQL editor prototype has been developed with a web interface providing various functionalities. Experimentation on scaling were performed, showing how sampling can speed up the computation time while keeping a satisfying quality. In addition, user experimentation was conducted with 70 participants: participants with access to extensions performed faster and adapted very well to tool. In the current context where more and more data is being stored and analyzed, such a tool can be useful, and could be integrated to data exploration solutions to improve them and give an help integrated to the querying language itself. Moreover, the solution is iterative and allows the user to modify an extension and to continue until she reaches what she was looking for. The extension tool is also a way to integrate knowledge on data, usually provided by data mining systems and tools, without leaving the context of DBMS. Many extensions of this work can be envisioned. It could be interesting to look at other SQL clauses to adapt the notion of extension to them, such as `group by` for example, to be able to automatically suggest grouping conditions. Finally, this work is also a contribution to bridge the gap between database techniques and machine learning techniques.

Appendix A. Experimentation Scenario

You're a new member of a post office, in charge of packages. When you're not at the front desk taking care of customers, you have access to data recorded about the packages sent from your post office. For simplification, we will focus on the packages leaving the post office to other destinations. Here is how the database was created :

```
CREATE TABLE Cities(
    city_ID DECIMAL,
    distance DECIMAL,
    PRIMARY KEY (city_ID)
)
CREATE TABLE Packages(
     package_ID DECIMAL,
     destination DECIMAL,
     length DECIMAL,
     width DECIMAL,
     height DECIMAL,
     weight DECIMAL,
     price DECIMAL,
     PRIMARY KEY (id_colis)
     FOREIGN KEY (destination)
          references Villes(id_ville)
)
```

Table Packages has one entry per package that left your post office. From the destination of a package, you can see how far it was sent, by joining tables *Packages* (11000 tuples) and *Cities* (30 tuples) on attributes *destination* and *city_ID*.

Questions are ordered from easiest to hardest :you should therefore answer them in the given order. First three questions are simple, while the others are voluntary more complex, and finding the required SQL query in questions 4 to 10 required more exploration.

Questions

Question 1 This first question is here so that you can get familiar with the data and the tools at your disposal. Please test the two tools (SQL

software and online form for answers) with the following query, that is a join between the two tables (Expected result size: 10 999 tuples):

```
Select *
From Packages, Cities
Where Packages.destination = Cities.city_ID
```

Question 2 Maximum size limit authorized for a package is 9000 grams. However, some exceed this limit without being detected. Give the query to obtain the ID of packages whose weight exceed this limit. (Expected result size: 73 tuples)

Question 3 What query can you write to obtain the average length of packages sent less than 100 kilometers from your post office ? (Expected result size: 1 tuple)

Question 4 A little bit interested by data analysis, a colleague of yours had, with a spreadsheet, visualized some curves from the database. By plotting packages prices against their height, he/she had noticed a group of packages very distinct and well separated from the others, which is presented on figure A.7, and circled in red. Can you find the query that returns all packages belonging to this group ?(Expected result size: 33 tuples)

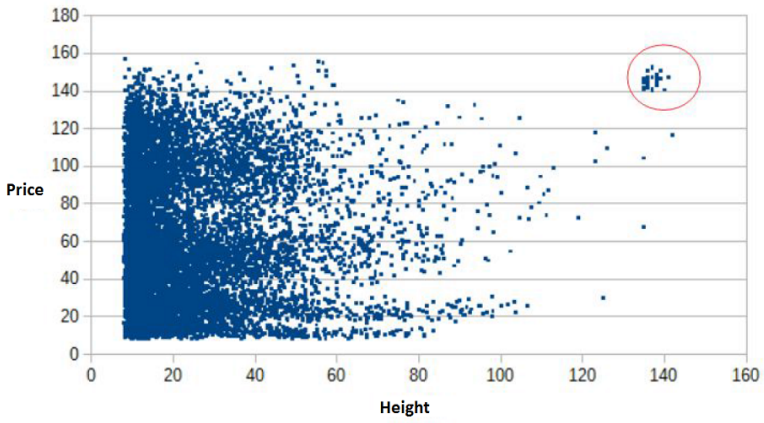


Figure A.7: Visualization for question 4

Question 5 According to some colleagues who've been working here for years, heaviest packages are the ones going to very distant destinations. The

intuition behind this is that sending a package far away is expansive, so customers put many things in one package to compensate. Can you identify packages that do not comply with this, i.e that are not heavy but are sent far away ? (Expected result size: 13 tuples)

Question 6 Once at the regional sorting center, packages go through a machine that automatically sorts them according to their destination. However, this machine is sometimes defective. Indeed, when a package is less than 480g, the machine does not always detect it, and an operator has to take it and process it manually. This phenomenon is marginal, but more likely to happen if in addition to its light weight, the packages is small regarding its length and width. On all packages registered in your database, 12 have caused a problem. Which query can identify those 12 packages ?

Question 7 Some packages are sent to a city that is very close to your post office, less than 10km away. Moreover, some are very light (less than 550g), and you wonder why people pay the post office to transport them while they would quite easily do it themselves. One of your colleagues has an hypothesis : maybe those packages are cumbersome and therefore hard to transport. Can you identify packages validating this hypothesis ? (Expected result size: 8 tuples)

Question 8 A customer arrives at the post office, because he needs the ID of a package he had send, but isn't able to find. In order to help him, he gives you a few informations: the package was light, less than 450g and its dimensions (mainly length and width) were surprisingly big in regard to its weight. Can you give the query returning such a package ? (Expected result size: 1 tuple)

Question 9 When working at the front desk, one of your colleagues made a mistakes on four on the packages he registered. Luckily, he remembers their length was above 140cm, and he therefore applied a special tarification, as those kind of packages are more complicated to deliver due to their size. But he applied the wrong tarification, and those packages have therefore an abnormally elevated price. Can you identify those packages ? (Expected result size: 4)

Question 10 At question 2, you showed that 73 packages are above the

weight limit. But your colleagues in charge of putting packages in the trucks say that a third of packages are really heavy, and require two employees to be lifted, in order to avoid back pains. Can you modify the query for question 2 in order to identify those packages ? (Expected result size : 3073 tuples)

References

[1] C. Mishra, N. Koudas, Interactive query refinement, in: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, ACM, 2009, pp. 862–873.

[2] M. J. Carey, D. Kossmann, On saying enough already! in sql, in: ACM SIGMOD Record, Vol. 26, ACM, 1997, pp. 219–230.

[3] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, Journal of computer and system sciences 66 (4) (2003) 614–656.

[4] M. Levene, G. Loizou, A Guided Tour of Relational Databases and Beyond, Springer-Verlag, London, UK, UK, 1999.

[5] A. K. Jain, Data clustering: 50 years beyond k-means, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 2008, pp. 3–4.

[6] Q. T. Tran, C.-Y. Chan, S. Parthasarathy, Query reverse engineering, The VLDB Journal 23 (5) (2014) 721–746. doi:10.1007/s00778-013-0349-3.
URL `http://dx.doi.org/10.1007/s00778-013-0349-3`

[7] L. Parida, N. Ramakrishnan, Redescription mining: Structure theory and algorithms, in: AAAI, Vol. 5, 2005, pp. 837–844.

[8] J. Han, Data Mining: Concepts and Techniques, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[9] J. H. Friedman, On bias, variance, 0/1loss, and the curse-of-dimensionality, Data mining and knowledge discovery 1 (1) (1997) 55–77.

[10] S. Lloyd, Least squares quantization in pcm, IEEE Trans. Inf. Theor. 28 (2) (2006) 129–137. doi:10.1109/TIT.1982.1056489.
URL `http://dx.doi.org/10.1109/TIT.1982.1056489`

[11] T. Sellam, M. Kersten, Cluster-driven navigation of the query space, IEEE Transactions on Knowledge and Data Engineering 28 (5) (2016) 1118–1131.

[12] Z. Huang, Extensions to the k-means algorithm for clustering large data sets with categorical values, Data mining and knowledge discovery 2 (3) (1998) 283–304.

[13] P. J. Rousseeuw, Silhouettes: a graphical aid to the interpretation and validation of cluster analysis, Journal of computational and applied mathematics 20 (1987) 53–65.

[14] J. Cumin, J.-M. Petit, V.-M. Scuturici, S. Surdu, Data exploration with sql using machine learning techniques, in: International Conference on Extending Database Technology-EDBT, 2017.

[15] L. Breiman, J. Friedman, C. J. Stone, R. A. Olshen, Classification and regression trees, CRC press, 1984.

[16] C.-C. Wu, Y.-L. Chen, Y.-H. Liu, X.-Y. Yang, Decision tree induction with a constrained number of leaf nodes, Applied Intelligence 45 (3) (2016) 673–685.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[18] L. Breiman, J. Friedman, R. Olshen, C. Stone, Classification and Regression Trees, Wadsworth and Brooks, Monterey, CA, 1984.

[19] M. Le Guilly, J.-M. Petit, V.-M. Scuturici, I. Ilyas, Explique: Interactive databases exploration with sql, in: 28th ACM International Conference on Information and Knowledge Management(CIKM '19), ACM, New York, NY, USA, 2019. doi:10.1145/3357384.335784.

[20] W. M. Rand, Objective criteria for the evaluation of clustering methods, Journal of the American Statistical association 66 (336) (1971) 846–850.

[21] W. G. Cochran, Relative accuracy of systematic and stratified random samples for a certain class of populations, The Annals of Mathematical Statistics (1946) 164–177.

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

[23] A. Motro, Vague: a user interface to relational databases that permits vague queries, ACM Transactions on Information Systems 6 (3) (1988) 187214. doi:10.1145/45945.48027.

[24] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, L. Novik, Discovering queries based on example tuples, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, ACM, New York, NY, USA, 2014, pp. 493–504. doi:10.1145/2588555.2593664.
URL http://doi.acm.org/10.1145/2588555.2593664

[25] A. Bonifati, R. Ciucanu, S. Staworko, Interactive join query inference with jim, Proc. VLDB Endow. 7 (13) (2014) 1541–1544. doi:10.14778/2733004.2733025.
URL http://dx.doi.org/10.14778/2733004.2733025

[26] D. M. L. Martins, G. Vossen, F. B. de Lima Neto, Learning database queries via intelligent semiotic machines, in: Computational Intelligence (LA-CCI), 2017 IEEE Latin American Conference on, IEEE, 2017, pp. 1–6.

[27] P. Utama, N. Weir, F. Basik, C. Binnig, U. Cetintemel, B. Hättasch, A. Ilkhechi, S. Ramaswamy, A. Usta, An end-to-end neural natural language interface for databases, arXiv preprint arXiv:1804.00401 (2018).

[28] D. Deutch, N. Frost, A. Gilad, T. Haimovich, Nlprovenans: natural language provenance for non-answers, Proceedings of the VLDB Endowment 11 (12) (2018) 1986–1989.

[29] O. Papaemmanouil, Y. Diao, K. Dimitriadou, L. Peng, Interactive data exploration via machine learning models., IEEE Data Eng. Bull. 39 (4) (2016) 38–49.

[30] K. Dimitriadou, O. Papaemmanouil, Y. Diao, AIDE: an active learning-based approach for interactive data exploration, IEEE Trans. Knowl. Data Eng. 28 (11) (2016) 2842–2856. doi:10.1109/TKDE.2016.2599168.
URL http://dx.doi.org/10.1109/TKDE.2016.2599168

[31] O. Papaemmanouil, Y. Diao, K. Dimitriadou, L. Peng, Interactive data exploration via machine learning models, IEEE Data Eng. Bull. 39 (4) (2016) 38–49.
URL http://sites.computer.org/debull/A16dec/p38.pdf

[32] A. Kashyap, V. Hristidis, M. Petropoulos, Facetor: cost-driven exploration of faceted query results, in: Proceedings of the 19th ACM international conference on Information and knowledge management, ACM, 2010, pp. 719–728.

[33] Y. Tzitzikas, N. Armenatzoglou, P. Papadakos, Flexplorer: A framework for providing faceted and dynamic taxonomy-based information exploration, in: Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on, IEEE, 2008, pp. 392–396.

[34] M. Drosou, E. Pitoura, Ymaldb: exploring relational databases via result-driven recommendations, The VLDB JournalThe International Journal on Very Large Data Bases 22 (6) (2013) 849–874.

[35] É. Fromont, H. Blockeel, J. Struyf, Integrating decision tree learning into inductive databases, in: International Workshop on Knowledge Discovery in Inductive Databases, Springer, 2006, pp. 81–96.

[36] N. U. Rehman, M. H. Scholl, Enabling decision tree classification in database systems through pre-computation, in: British National Conference on Databases, Springer, 2010, pp. 118–121.

[37] B. Liu, Y. Xia, P. S. Yu, Clustering through decision tree construction, in: Proceedings of the ninth international conference on Information and knowledge management, ACM, 2000, pp. 20–29.

[38] M. Zhang, H. Elmeleegy, C. M. Procopiuc, D. Srivastava, Reverse engineering complex join queries, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, 2013, pp. 809–820.

[39] S. Chaudhuri, Data mining and database systems: Where is the intersection?, Data Engineering Bulletin 21 (1998).

[40] B. Zou, X. Ma, B. Kemme, G. Newton, D. Precup, Data Mining Using Relational Database Management Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 657–667. doi:10.1007/11731139_75.
URL http://dx.doi.org/10.1007/11731139_75

[41] H. Wang, C. Zaniolo, C. R. Luo, Atlas: A small but complete sql extension for data mining and data streams, in: Proceedings of the 29th international conference on Very large data bases-Volume 29, VLDB Endowment, 2003, pp. 1113–1116.