Identifying User Interests within the Data Space – a Case Study with SkyServer

Hoang Vu Nguyen[○] Klemens Böhm[○] Florian Becker[○]
Bertrand Goldman[◇] Georg Hinkel[○] Emmanuel Müller[○●]

[○] Karlsruhe Institute of Technology (KIT), Germany
{hoang.nguyen, klemens.boehm, georg.hinkel, emmanuel.mueller}@kit.edu and florian.becker@student.kit.edu

[◇] Max Planck Institute for Astronomy, Germany
goldman@mpia.de

[●] University of Antwerp, Belgium
emmanuel.mueller@ua.ac.be

ABSTRACT

Many scientific databases nowadays are publicly available for querying and advanced data analytics. One prominent example is the Sloan Digital Sky Survey (SDSS)—SkyServer, which offers data to astronomers, scientists, and the general public. For such data it is important to understand the public focus, and trending research directions on the subject described by the database, i.e., astronomy in the case of SkyServer. With a large user base, it is worthwhile to identify the areas of the data space that are of interest to users.

In this paper, we study the problem of extracting and analyzing *access areas* of user queries, by analyzing the query logs of the database. To our knowledge, both the concept of access areas and how to extract them have not been studied before. We address this by first proposing a novel notion of access area, which is independent of any specific database state. It allows the detection of interesting areas within the data space, regardless if they already exist in the database content. Second, we present a detailed mapping of our notion to different query types. Using our mapping on the SkyServer query log, we obtain a transformed data set. Third, we aggregate similar overlapping queries by DBSCAN and gain an abstraction from the raw query log. Finally, we arrive at clusters of access areas that are interesting from the perspective of an astronomer. These clusters occupy only a small fraction (in some cases less than 1%) of the data space and contain queries issued by many users. Some frequently accessed areas even do not exist in the space spanned by available objects.

1. INTRODUCTION

Nowadays, many scientific databases are made publicly available to reach a large community of users. Popular examples are SkyServer from astronomy and GBIF from biosystematics. With a large user base, it is of great benefit to identify the parts of the data space that many users are interested in. This data space is formed by the database schema, which defines the domain of each column and their relationships. Note that the data space is not constrained to the actual database content. For instance, Figure 1(a) plots the subspace formed by two columns *plate* and *mjd* of relation *SpecObjAll* of SkyServer. One can see that the database content does not span the whole data space, leaving an *empty area*, i.e., part of the data space containing no data object.

A good indicator of the interestingness of a (sub)area of the data space is the frequency it is referred to in user queries. Identifying common interests of many users is useful for traditional applications such as performance tuning and query personalization [4, 17, 22]. However, it also is important for learning the database usage, which tends to represent the focus of the respective scientific community. In fact, our study of the SkyServer query log reveals that the interests of users in the data space often correspond to a small part of the database content. Furthermore, users even are interested in *empty* areas of the data space. Inspecting data objects queries actually have retrieved does not give rise to such insights. The following example is an excerpt of our results.

EXAMPLE 1. *As shown in Figure 1(a), the content of relation* SpecObjAll *is within the area*

$$(266 \leq \mathit{SpecObjAll.mjd} \leq 5141) \wedge (51578 \leq \mathit{SpecObjAll.plate} \leq 55752)$$

*of the subspace (*SpecObjAll.plate, SpecObjAll.mjd*) of the data space. An area returned by our algorithm that is accessed by* $18,904$ *queries is*

$$(296 \leq \mathit{SpecObjAll.plate} \leq 3200) \wedge (51578 \leq \mathit{SpecObjAll.mjd} \leq 52178) \quad ,$$

i.e., a small part of the content of SpecObjAll.

*In Figure 1(b), the area of the subspace (*PhotoObjAll.ra, PhotoObjAll.dec*) that queries refer to spans not only its database content, but also its* empty area. *The number of such 'empty area' queries is significant, see Table 1 in the evaluation.*

Figure 1(c) depicts a scenario similar to that of Figure 1(b). However, here the empty areas of the subspace accessed by queries are not contiguous and are larger than the part occupied by the database content.

The access areas found can be used in various ways: They could help funding agencies to align the spending of their resources with community interests. In the context of SkyServer, priority could be given to projects exploring the area of the sky many astronomers/ scientists/students/the public are interested in. Further, they could help researchers to identify 'gaps' in the current scientific knowledge systematically and to select new, good research topics that are up to what many people are targeting.

The problem studied in this paper is to identify areas of interest to many users within the data space, at the right level of abstraction, from the query log. Our approach rests on the following pillars:

- The database offers a flexible query interface such as SQL. Such databases and means to query it flexibly are fundamentally important in many scientific domains.
- The database serves a large number of users, i.e., a large community. This ensures the value of the common interests extracted.
- Most users do not have personal contact with the database owner. Hence, he/she does not have an objective picture of what users really are after. In other words, the anecdotical evidence that he/she may have is not necessarily representative of the interests of the user group as a whole.
- Most users of a scientific database are knowledgeable on its domain, e.g., by working with domain experts with whom they have collaboration. This implies that most queries issued by users are meaningful. Furthermore, such databases limit the number of queries each user is allowed to issue within a window of time. This makes queries well-designed, but also hinders us to re-issue a large number of queries to collect statistics.

In such a setting, the problem we study can be more precisely phrased as: *Given the query log containing SQL statements issued by users, how to extract their intents, i.e., the areas of the data space many users wanted to access, without accessing (re-querying) the database?*

We see three main challenges in the way of solving this problem. First, we need to come up with a formal definition of access area, i.e., the area of the data space that a query refers to. The definition needs to be sufficiently abstract, so that it is not confined to a specific data model or even a certain database schema. Further, it must not be confined to the database content and enable the discovery of empty areas of the data space many users care about. Second, given an abstract definition of access area, we need to come up with a mapping of queries to their access areas, or, in other words, a realization of the definition on all query types actually occurring. This is far from obvious, due to the expressiveness of modern query languages. In fact, it turns out that in some cases the mapping is overly complex and sophisticated analysis is required. Third, we need a procedure to aggregate the access areas of a large set of queries. Clustering seems promising, but to do so, we need a distance measure. Such a distance measure needs to be defined, since existing ones for queries focus on their structure [4]. Here in turn, the focus has to be on the content (i.e., access areas) and the similarity of overlapping areas for a meaningful abstraction of a set of queries.

This paper represents our solution to each of these challenges. In particular, we introduce the novel concept of access area, which is independent of any specific data model and any database state. Being independent of the database content brings a performance gain compared to actually rerunning the queries. In addition, our notion of access area lets us achieve our important goal: We discover areas of the data space that are of interest to users, irrespective of the number of data objects falling into these areas. Second, we provide a mapping of our notion to all query types occurring in the SkyServer query log, i.e., we enable extraction of access areas in practice. We also show that extracting access areas is not straightforward for queries with joins, aggregate queries, and nested queries. Third, we exploit query overlap for the aggregation of access areas of a large set of queries.

At this point, our main interest is on the application side, i.e., to find out whether our approach makes sense for domain experts (it does), whether the results obtained so far are plausible (mostly yes), and whether they allow for new insights regarding user interests (not so much at this point, but there are promising starting points for refinements). We have learned this from a case study on SkyServer and interviewing both a person responsible for SkyServer and an 'independent' astronomer. In the study, we extract access areas from the SkyServer query log and cluster the transformed data with DBSCAN. The clustered access areas found occupy only a small fraction (in some cases less than 1%) of the data space and are accessed by many users. We also detect access areas which do not even contain any data objects. Such areas can be rather large (see Figure 1(c)). Finally, our astronomer does not only deem our approach helpful for the owner of the data, but also for users.

While our focus so far has been on SkyServer, our concepts are applicable to any database, i.e., no confinement to RDBMSs. Further, since no SkyServer-specific features have been hard-coded in our implementation, it is applicable to any SQL database and facilitates any future extension to cope with databases in other domains. We repeat that our concern with this paper is to propose an approach that leads to interesting results. Tuning the approach, e.g., by experimenting with other distance functions or clustering algorithms systematically, is beyond our current scope.

Paper outline: In Section 2, we provide our definition of acess area. In Section 3, we review related work. In Section 4, we describe our implementation. In Section 5, we present our distance function. Section 6 features our case study; Section 7 concludes.

2. ACCESS AREA

The access area of a query captures the area of the data space that the user is interested in. We now formalize this concept. To ease our presentation, we first introduce some notation. Then we discuss two straightforward ways to define access areas and point out their drawbacks. Finally, we propose our notion of access area.

2.1 Preliminaries

We consider a relational database $\mathbf{DB}$ which consists of multiple relations; each relation has several columns.

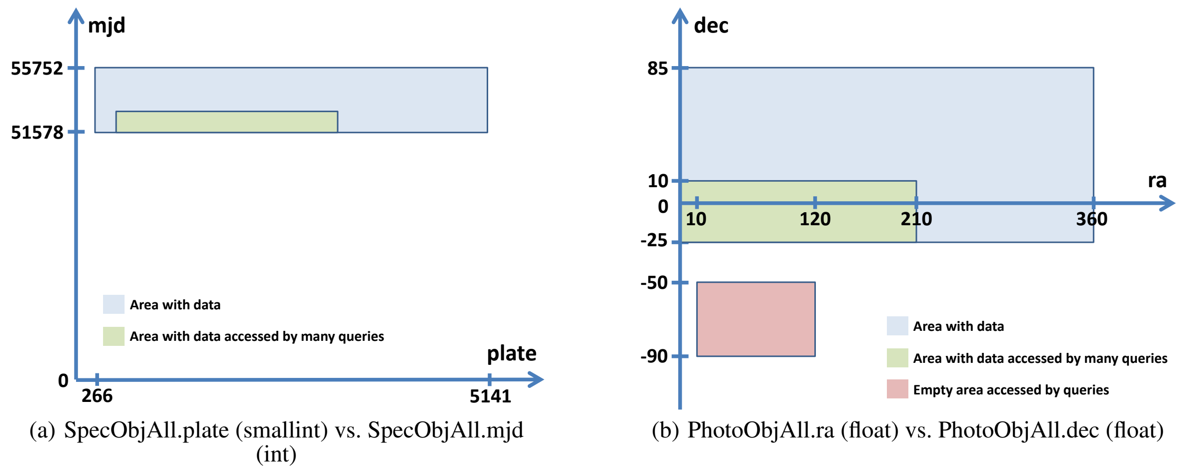
Data spaces.

The data space of a relation $R \in \mathbf{DB}$ which consists of columns $a_1, \ldots, a_t$ is defined as

$$space(R) = dom(a_1) \times \ldots \times dom(a_t)$$

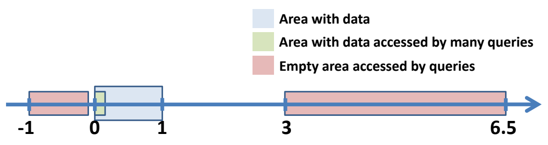
where $dom(a_i)$ is the domain of column a_i. In other words, the data space of R is the Cartesian product of the domains of its columns. Note that the data space of R is not confined to the database content. Further, when $t = 1$, i.e., R contains one column only, $space(R) = space(a_1) = dom(a_1)$. In the rest of this paper, unless specified otherwise, we use R to denote both the relation R and $space(R)$.

The data space of $\mathbf{DB}$ is defined to be the Cartesian product of all of its relations. The data space of each relation is a *subspace* of the data space of $\mathbf{DB}$.



(a) SpecObjAll.plate (smallint) vs. SpecObjAll.mjd (int)

(b) PhotoObjAll.ra (float) vs. PhotoObjAll.dec (float)



(c) Photoz.z (real)

Figure 1: Some access areas extracted from SkyServer query log.

Area of data content and empty area of a data space.

Consider again relation R with columns $a_1, \ldots, a_t$. For each column a_i, if a_i is numerical, we let $content(a_i)$ be the minimum bounding box of its data values. If a_i is instead categorical, we let $content(a_i)$ be the set of values of a_i.

The area of data content of R is defined as

$$content(R) = content(a_1) \times \ldots \times content(a_t) \quad ,$$

i.e., the minimum bounding hyper-rectangle of the data content.

The empty area of $space(R)$ is $empty(R) = space(R) \setminus content(R)$. Roughly speaking, $empty(R)$ is the part of $space(R)$ which is not occupied by any database object. Note that this is a conservative way of estimating $empty(R)$. A more stringent notion could help to identify larger empty areas accessed by users.

Atomic predicates.

Atomic predicates are those without deeper propositional structure. Though having many forms, we focus on atomic predicates having the form of "$a\ \theta\ c$" where a is a database column, c is a constant, and θ is either $<, \leq, =, >, \geq$, or $<>$. We name this type of predicate as *column-constant* atomic predicate.

Queries.

Consider a query **q** in the log, which is issued against a state $\mathcal{T}$ of database $\mathbf{DB}$. Typically, **q** consists of SELECT, FROM, and possibly WHERE, GROUP BY, HAVING and ORDER BY clauses. The ORDER BY clause is not relevant for our purpose since it does not influence which parts of the data space actually are accessed. Thus, from now on, we exclude the ORDER BY clause.

Assume that **q** accesses relations $\mathcal{R} = \{R_1, \ldots, R_N\}$ where $R_i \neq R_j$ if $i \neq j$. Note that this excludes self-joins, which do not occur in the SkyServer query log that we considered. This type of join typically results in different aliases of the same relation and causes unreliable comparison of similarity between queries [4].

Each relation $R_i \in \mathcal{R}$ can appear either in the FROM clause or in any other clause, possibly in embedded subqueries. The area of $\mathbf{DB}$ accessed by **q** is a subset of $R_1 \times \cdots \times R_N$, which is called the universal relation of **q**:

DEFINITION 1. *The universal relation of **q** is defined as:*

$$\mathcal{U} = R_1 \times \cdots \times R_N \quad .$$

The set of tuples of $\mathcal{U}$ at state $\mathcal{T}$ of $\mathbf{DB}$ is denoted as $(\mathcal{U}, \mathcal{T})$.

Each output column of **q** may or may not belong to the relations accessed, since new output columns may be created, e.g., columns holding constant values. We denote the set of columns appearing in the WHERE clause of **q** as $\mathcal{A}_W$, the ones in the GROUP BY clause as $\mathcal{A}_G$, the ones in the HAVING clause as $\mathcal{A}_H$, and the ones in any (nested query of) other clauses as $\mathcal{A}_S$. We note that $\mathcal{A}_W$, $\mathcal{A}_G$, and $\mathcal{A}_H$ may be empty. We define $\mathcal{A} = \mathcal{A}_W \cup \mathcal{A}_G \cup \mathcal{A}_H \cup \mathcal{A}_S$.

The WHERE clause can contain one or more atomic predicates on the columns of some relation(s) in $\mathcal{R}$. Likewise, the HAVING clause can contain one or more atomic predicates on the columns of some relation(s) in $\mathcal{R}$, typically in combination with aggregate functions like SUM, AVG, etc. Additional predicates can appear in (nested queries of) any other clause. Together, all these predicates and their connections form a constraint on $\mathcal{U}$, in the form of a Boolean expression. We refer to this Boolean expression as $\mathcal{P}$. Note that $\mathcal{P}$ is independent of any database state. As shown in Section 4, it is not always easy to extract $\mathcal{P}$ from **q**. Instead, this depends on the type of **q** as well as on the definition of access area. For now, we simply use $\mathcal{P}$ for the sake of exposition. We have:

DEFINITION 2. *The result set of **q** at state $\mathcal{T}$ is the tuples of $(\mathcal{U}, \mathcal{T})$ which satisfy $\mathcal{P}$. We denote it as $(\mathcal{U}, \mathcal{T})_{\mathcal{P}}$.*

Following Definition 2, we have $(\mathcal{U}, \mathcal{T})_{\text{TRUE}} = (\mathcal{U}, \mathcal{T})$.

2.2 Naïve Definitions of Access Area

At first sight, one can define the access area of a query **q** to be either (a) the area of the data space covered by its result set, or (b) the part of $\mathbf{DB}$ accessed during the execution of **q**, as follows.

Option (a).

The access area of **q** is its result set at state $\mathcal{T}$, i.e., $(\mathcal{U}, \mathcal{T})_{\mathcal{P}}$. Under this scheme, an access area can be empty when the user is interested in an area where no object exists. Using this definition, we could define the access area of **q** as the minimum bounding box of $(\mathcal{U}, \mathcal{T})_{\mathcal{P}}$. Typically, $(\mathcal{U}, \mathcal{T})_{\mathcal{P}}$ is not available in the query log. This means that we have to re-issue **q** against the database, most likely at a different state $\mathcal{T}'$. So this definition suffers from two drawbacks. First, we may lose important information on the constraint $\mathcal{P}$ that the user had defined, as two very different queries can share the same result set (or minimum bounding box) independently of their constraints, i.e., their original intents. Second, if $\mathcal{T}' \neq \mathcal{T}$, it may be the case that $(\mathcal{U}, \mathcal{T}')_{\mathcal{P}} \neq (\mathcal{U}, \mathcal{T})_{\mathcal{P}}$. Thus, this definition is not meaningful for our purpose.

Option (b).

The access area of **q** is the part of $\mathbf{DB}$ at state $\mathcal{T}$ which has been accessed during the execution of **q**. In general, the query engine determines the part of the database to be accessed. To accomplish this, it relies on different statistics, e.g., query workload, network statistics, to decide on an execution plan for the query. In consequence, this definition of access areas leaves them dependent on factors that do not pertain to queries themselves. This is undesirable for capturing intents of users. Further, it is also difficult to impossible to compute access areas with an off-the-shelf commercial DBMS.

2.3 Our Definition of Access Area

DEFINITION 3. *Let a database state $\mathcal{T}$ of $\mathbf{DB}$ allowed by the database schema be given. A tuple $t \in \mathcal{U}$ is said to **influence** the result set $(\mathcal{U}, \mathcal{T})_{\mathcal{P}}$ of **q** iff:*

$$(\mathcal{U} \setminus \{t\}, \mathcal{T})_{\mathcal{P}} \neq (\mathcal{U}, \mathcal{T})_{\mathcal{P}} \quad .$$

That is, if t is removed from $\mathcal{U}$, the result set of **q** at state $\mathcal{T}$ will change. Our definition of access area now is as follows:

DEFINITION 4. *The access area of **q** is:*

$$\{t \in \mathcal{U} : \exists \mathcal{T} \text{ allowed by } \mathbf{DB} \text{ s.t. } t \text{ influences } (\mathcal{U}, \mathcal{T})_{\mathcal{P}}\} \quad .$$

That is, the access area of **q** is given by the set of all tuples contained in the universal relation $\mathcal{U}$ that influence the result set of **q** in **some** state allowed by the database schema. Following Definition 4, the access area of a query **q** represents the part of the data space the user is interested in, without limiting it to a certain state. For instance, let us consider the following example query:

```
SELECT *
FROM T
WHERE u BETWEEN 1 AND 8
```

According to Definition 4, the access area of this query is $\sigma_{u \geq 1 \wedge u \leq 8}(T)$, even if $u \notin [1, 8]$ for all tuples of relation T in its current state. This is because, for any tuple that satisfies $1 \leq u \leq 8$, a database state can exist, allowed by the schema, such that the tuple influences the result.

Therefore, our definition of access area copes with queries that do not return any row as well as with those that resulted in execution errors, e.g., *"Maximum 60 queries allowed per minute"* or *"limit is top 500000"* in the SkyServer scenario. This is crucial as it makes sense to capture what the users intended to access, independent of the actual database content and load constraints.

2.4 Extraction of Access Areas

To extract the access area of **q**, we have to obtain $\mathcal{U}$ and $\mathcal{P}$. While extracting $\mathcal{U}$ is rather simple, extracting $\mathcal{P}$ is more intricate. In fact, it is not always possible to extract $\mathcal{P}$ exactly. When this is the case, we derive a Boolean expression in conjunctive normal form which approximates $\mathcal{P}$. With the derivation of $\mathcal{P}$ or its approximation, we typically transform **q** to an **intermediate** format. In particular, concurrently with identifying the access area of **q**, we transform **q** to a query **q'** as follows:

SELECT $*$
FROM $R_1, R_2, \ldots, R_N$
WHERE
$F(p_1(a_{1,1}, a_{2,1}, \ldots, a_{T_1,1}), \ldots, p_K(a_{1,K}, a_{2,K}, \ldots, a_{T_K,K}))$

where

- $\{a_{1,1}, \ldots, a_{T_K,K}\} \subset \mathcal{A}$,
- p_i for every $i \in [1, K]$ being an atomic predicate defined on columns $\{a_{1,i}, a_{2,i}, \ldots, a_{T_i,i}\}$, and *derived* from $\mathcal{P}$,
- F being a *conjunctive normal form* of the atomic predicates $\{p_1, \ldots, p_K\}$, i.e., a conjunction of disjunctions. For instance, assume that $K = 4$, $F(p_1, p_2, p_3, p_4)$ could be $(p_1 \vee p_2) \wedge (p_3 \vee p_4)$.

We note that the WHERE clause is optional and can be empty.

With the above transformation, the access area of query **q'** is the one of **q**, which is $\sigma_{F(p_1,\ldots,p_K)}(R_1 \times R_2 \times \cdots \times R_N)$, or $\sigma_{F(p_1,\ldots,p_K)}(\mathcal{U})$.

For illustration, the following query is already in intermediate format:

```
SELECT *
FROM T
WHERE (T.u <= 5 OR T.u >= 10) AND T.v <= 5
```

So no transformation is required and we can obtain its exact access area easily.

However, in practice, an exact transformation is not always straightforward for most query types, especially nested queries in combination with (NOT) IN, (NOT) EXISTS, (NOT) ALL and (NOT) ANY. Instead, sophisticated analysis is required. More details are in Section 4. Having introduced our notion of access area, we discuss related work in the next section and point out why it is not suited to address the problem.

3. RELATED WORK

Processing and extracting useful information from SQL queries has been studied for some time. In this paper, we divide related work into the following categories: extracting information from queries, processing query logs, and distance measures for queries.

3.1 Extracting Information from Queries

Ceri and Gottlob [6] studied transforming SQL queries into relational algebra. Their goal is to preserve the constraints defined by typical SQL structures such as EXISTS, IN, and aggregate functions. However, since they did not focus on extracting access areas, they have not introduced methods to convert complex Boolean expressions, such as those involved in set operations (with IN, ANY, etc.), to simple ones with atomic predicates. Such simple Boolean expressions in turn are required to constrain the data space, and hence, for the extraction of access areas.

Parsing, relaxing and rewriting of queries are investigated in [9, 14, 18, 21]. Their primary goal is to either point out logical flaws in query sub-expressions, rewrite queries into an optimized, more declarative form that avoids empty results and improves query performance. Thus, they do not address the problem studied here.

A more intuitive way of representing queries is reported in [11]. The meaning (or intent) of queries is captured by an UML-like notation. This notation breaks down the query structure to show relationships between different fragments. Yet, the proposed method does not have a mechanism to further reduce complex fragments, such as EXISTS, to simple ones.

There also is related work to information extraction of queries from fields such as Natural Language Processing (NLP), Information Retrieval (IR), and Machine Learning (ML). For example, [15] and [19] propose to transform SQL queries to a different format—the one of natural language. In particular, queries are represented by graphs using a query template mechanism. The critical elements of SQL queries are extracted, and different strategies are presented to construct textual query descriptions from these elements. However, these studies also do not target at simplifying Boolean expressions to facilitate the extraction of access areas.

3.2 Processing Query Logs

Due to the vast amount of data produced by search engines and the pervasive tracking of user click-streams, researchers have studied query log processing to some extent. For instance, [17] and [22] aim at furthering our understanding of the behavior of users through their information-seeking activities. These articles demonstrate why query log processing is useful for data analytics. In more details, [17] transforms web logs into a format suitable for importing into databases. In addition, it builds a data warehouse from these cleaned and structured log files and provides an ad-hoc tool for analytic queries on this warehouse. [22] explores different statistics, which can be drawn from query logs such as query popularity, term popularity, average query length, and distance between repetitions of queries. [3, 4] on the other hand target OLAP logs with the objective to compare different users sessions.

Singh et al. [23] give a detailed analysis of the first five years since SDSS SkyServer went on-line. They clean and normalize web and SQL log files over several months, resulting in data structures including IP Name, Sessions, and Templates (skeleton SQL templates). They further compare SQL queries using fragments, N-grams, and the Jaccard Coefficient, and categorized the queries into those issued by bots or by mortals.

QueRIE, a query recommendation system [2, 8, 7, 20], is designed to work directly with SkyServer query logs. It uses two different approaches to achieve high accuracy in recommending new and interesting queries to users. The first approach extracts the most important SQL query fragments, while the second one uses the tuples a query retrieves.

SDSS Log Viewer [26] visualizes the SQL queries from the log files. To achieve the visualization, the author develops a process to transform the SkyServer SQL log files into a tabular format that can be stored in a database. Each query is tokenized in order to apply a visual encoding scheme and to extract the critical fragments. Depending on these fragments, queries are classified into four categories representing the type of "sky area" a query accesses: Rectangular Sky Area, Circular Sky Area, Single Point/Object, and others. Besides this, depending on the intention of the user, the author creates three categories for the major sets of queries: Scan Queries, Search Queries, and Retrieve Queries.

All of the above approaches do not solve the problem we study as they (a) neglect the notion of access area in each query, and (b) lack mechanisms to map queries to access areas.

3.3 Distance Measures for Queries

Existing distance measures for queries model them as either strings [25], feature vectors [1, 2, 12], sets of fragments [13], or graphs [24]. With all these representations, clustering is expected to be feasible. But these clusters either do not represent specific parts of the data space, or the mapping would need to be specified in the first place. In contrast to these, we focus on a similarity notion that allows us to aggregate a set of overlapping access areas.

4. REALIZATION

Different types of queries need different types of predicate extraction, i.e., different mappings to their access areas. Further, as we will explain in this section, for some types, non-trivial analysis is required to extract their access areas. As the SQL grammar has a high complexity, and as many variations exist between different database management systems, it lies beyond the scope of this paper to cover every possible valid SQL statement. Instead, we focus on the log file of one large database and extract the access areas from the statements in this log file. We consider $12,375,426$ SkyServer queries issued by users from 127 countries, starting from April 2012. We note that since the number of queries considered is large and their origins are diverse, we do not expect any major gaps when working with another public database or returning to SkyServer in the future. Further, our model can always be extended later to include yet unhandled types of queries or types of predicates, schema specific functions, as well as different SQL dialects. It is also possible to extract the information from an incoming stream of logged queries, to detect changes in this data stream and to notify the system operator about the occurrence of new predicates and query types.

For our system implementation, we use the log file from the 9^{th} data release of SDSS, which has been the latest release when we started this project. Currently, we classify the queries logged into several categories to facilitate processing. Nevertheless, we apply the same procedure for any query type. That is, following Section 2.4, before extracting the access area of a query, we transform it into a query of the intermediate format, if necessary. Then we process the transformed query to obtain the access area. The details are as follows.

4.1 Simple Queries

Queries belonging to this category are those without (a) joins, (b) GROUP BY and HAVING clauses, and (c) nested subqueries. In other words, each query **q** of this type either already is of the intermediate format or can be straightforwardly converted to the intermediate format (which involves in converting its constraint $\mathcal{P}$ to a conjunctive normal form). For each simple query, since its predicates can be extracted exactly, we can easily obtain its exact access area. For example, the following query

```
SELECT u
FROM T
WHERE u >= 1 AND u <= 8 AND s > 5
```

is a simple query with access area $\sigma_{u\geq 1 \wedge u\leq 8 \wedge s>5}(T)$.

From an implementation point of view, we need special handling of queries containing specific operators, namely BETWEEN and NOT. In particular, for each predicate with a BETWEEN operator, we need to derive two new predicates to replace the original one. For example, $T.u$ *BETWEEN* 5 *AND* 10 is converted to

$T.u \geq 5 \wedge T.u \leq 10$. For predicates containing the NOT operator, we transform them by inverting the respective predicate. For example, *NOT* $(T.u > 5 \wedge T.v \leq 10)$ becomes $T.u \leq 5 \vee T.v > 10$.

4.2 Join Queries

Most types of JOINs can be converted simply by keeping the relation and pushing any join condition to the WHERE clause (CROSS JOIN, INNER JOIN, EQUI JOIN, NATURAL JOIN). Other types need further consideration, as follows.

EXAMPLE 2. *Consider a query **q** with FULL OUTER JOIN:*

```
SELECT *
FROM T FULL OUTER JOIN S ON (T.u = S.u)
```

Here, the relations involved are T and S, i.e., $\mathcal{U} = T \times S$. Regarding the constraint on $\mathcal{U}$, we observe that FULL OUTER JOIN keeps all tuples of both relations, even if there is no match for $T.u = S.u$. Thus, any tuple in $T \times S$ can influence the result set of the original query. That is, there is no constraint on $\mathcal{U}$. Hence, we convert the query to:

```
SELECT *
FROM T, S
```

The access area then is $\sigma(T \times S)$.

From Example 2, we see that identifying $\mathcal{P}$ goes beyond simply extracting the predicates as-is.

EXAMPLE 3. *Consider a query **q** with RIGHT OUTER JOIN:*

```
SELECT *
FROM T RIGHT OUTER JOIN S ON (T.u = S.u)
```

Again, we have $\mathcal{U} = T \times S$. However, identifying $\mathcal{P}$ is more complex. We observe that this query returns all tuples in S, together with those tuples from T which match at least one tuple in S. That is, it is equivalent to:

```
SELECT *
FROM T, S
WHERE T.u IN (SELECT S.u FROM S)
```

The above query is nested, and we have a special procedure to handle it, which comes in Section 4.4. Queries with LEFT OUTER JOIN are handled analogously.

4.3 Aggregate Queries

There is a variety of aggregate queries in practice, too many to be covered fully. Thus, we confine our study to aggregate queries that can be found in the SkyServer query log. Such queries have the following form:

```
SELECT *
FROM [...]
WHERE [...]
GROUP BY [...]
HAVING AGG(a) [< | ≤ | = | > | ≥ | <>] c
```

That is, the HAVING clause is of the form $AGG(a)\ \theta\ c$ where a is a column and θ is either $<$, $\leq$, $=$, $>$, $\geq$, or $<>$. Further, c is a constant. The FROM clause can consist of any number of relations. In addition, the WHERE clause is a conjunctive normal form of atomic predicates. The GROUP BY clause in turn can consist of any combination of columns. Here, GROUP BY and WHERE clauses are optional. We consider the aggregate functions SUM(), COUNT(), MIN(), MAX(), and AVG() in our implementation, though we note that MAX() does not appear in the log.

Overall, we find an exact transformation that preserves access areas for queries of the above format. As a representative, we illustrate our handling of such queries for the SUM function. To this end, we observe that there are several scenarios; some of which are shown below. The others are in [5].

LEMMA 1. *Consider the following query:*

```
SELECT T.u, SUM(T.v)
FROM T
GROUP BY T.u
HAVING SUM(T.v) > c
```

where c is a constant. Assume that $dom(T.v) = [inf, supp]$ where $supp$ could be $+\infty$, and inf could be $-\infty$. We have:

- *If $supp > 0$, the access area is T.*
- *If $supp \leq 0 \wedge c > supp$, the access area is $\emptyset$.*
- *If $supp \leq 0 \wedge c \in dom(T.v)$, the access area is $\sigma_{T.v>c}(T)$.*
- *If $supp \leq 0 \wedge c < inf$, the access area is T.*

PROOF. Consider an arbitrary tuple $t \in T$ (i.e., t is a tuple in the data space of T).

Case 1: $supp > 0$. Let k be an integer such that

$$k > \left\lceil \frac{2(c - t.v)}{supp + \max\{inf, 0\}} \right\rceil \quad .$$

Consider a database state in which T contains t and k other tuples $\{t'_1, \ldots, t'_k\}$; each tuple t'_i satisfies that $t'_i.u = t.u$ and $t'_i.v = \frac{supp+\max\{inf,0\}}{2}$. Since $t'_i.v \in dom(T.v)$, this state is allowed by the database schema. We have: $t.v + \sum_{i=1}^{k} t'_i.v > c$. Thus, t influences the result set, i.e., the access area is T.

Case 2: $supp \leq 0$. We have the following cases:

- $c > supp$: For every $x' \in dom(T.v)$, it holds that: $t.v + x' \leq t.v < c$. This implies that t can never be part of the access area. Thus, the access area is $\emptyset$.
- $c \in dom(T.v)$: If $t.v > c$, we construct a database state in which T contains t only, which conforms to the database schema. Further, t influences the result set. In contrast, if $t.v \leq c$, we can deduce that t cannot influence the result set. Therefore, the access area is $\sigma_{T.v>c}(T)$.
- $c < inf$: The access area is T.

This concludes our proof. □

In the following lemmas, for simplicity, we assume that the domain of each column involved is large enough such that with respect to its data type, it can be considered as $(-\infty, +\infty)$. This holds in general since queries tend to not have predicates containing values near the bounds of domains.

LEMMA 2. *Consider the following query:*

```
SELECT T.u, SUM(T.v)
FROM T
WHERE T.v < c1
GROUP BY T.u
HAVING SUM(T.v) > c2
```

where c_1 and c_2 are constants. We have:

- *If $c_1 > 0$, the access area is $\sigma_{T.u<c_1}(T)$.*
- *If $c_1 \leq 0$ and $c_2 \geq 0$, the access area is $\emptyset$.*
- *If $c_1 \leq 0$ and $c_2 < 0$: If $c_2 < c_1$, the access area is $\sigma_{T.u<c_1 \wedge T.u>c_2}(T)$. Otherwise, it is $\emptyset$.*

PROOF. Consider an arbitrary tuple $t \in T$ (i.e., t is a tuple in the data space of T). If $t.v \geq c_1$, $t.v$ is not part of the access area. Hence, we will only consider the case where $t.v < c_1$.

$c_1 > 0$: Let k be an integer such that $k > \left\lceil \frac{2(c_2 - t.v)}{c_1} \right\rceil$. Consider a database state in which T contains t and k other tuples $\{t'_1, \ldots, t'_k\}$ where $t'_i.u = t.u$ and $t'_i.v = \frac{c_1}{2}$. Since $\frac{c_1}{2}$ is a valid value of $T.v$, this state is allowed by the database schema. We have: $t.v + \sum_{i=1}^{k} t'_i.v > c_2$. Thus, the access area is: $\sigma_{T.v<c_1}(T)$.

$c_1 \leq 0$ and $c_2 \geq 0$: For every $x' < c_1$, it holds that: $t.v + x' \leq t.v < c_1 \leq c_2$. This implies that t can never be part of the access area. Thus, the access area is $\emptyset$.

$c_1 \leq 0$ and $c_2 < 0$: Consider an arbitrary $x' < c_1$. If $t.v \leq c_2$, we have: $t.v + x' < t.v \leq c_2$, i.e., t is not part of the access area. As a result, if $c_2 < c_1$, the access area is: $\sigma_{T.u<c_1 \wedge T.u>c_2}(T)$. Otherwise, it is $\emptyset$. □

LEMMA 3. *Consider the following query:*

```
SELECT T.u, SUM(T.v)
FROM T
WHERE T.v > c1
GROUP BY T.u
HAVING SUM(T.v) > c2
```

where c_1 and c_2 are constants. The access area of this query is $\sigma_{T.u>c_1}(T)$.

PROOF. Consider an arbitrary tuple $t \in T$ (i.e., t is a tuple in the data space of T) where $t.v > c_1$.

$c_1 > 0$: Let $k > \left\lceil \frac{c_2 - t.v}{c_1} \right\rceil$. We have: $k \cdot c_1 > c_2$. Analogously to Lemma 2, the access area is: $\sigma_{T.u>c_1}(T)$.

$c_1 \leq 0$: Let $k > \lceil c_2 - t.v \rceil$. Consider a database state in which T contains t and k other tuples $\{t'_1, \ldots, t'_k\}$ where $t'.u = t.u$ and $t'.v = 1$. This state is allowed by a database schema. In addition, $t.v + \sum_{i=1}^{k} t'_i.v > c_2$. Thus, the access area still is: $\sigma_{T.u>c_1}(T)$. □

In our implementation, we have covered all cases for the SUM function. For each query with $\text{SUM}(a)\ \theta\ c$ in the HAVING clause, we check if a belongs to some relation in the FROM clause. If it does not, we ignore it. Otherwise, we apply special mappings. The above cases are examples of such mappings. The detailed handling of other cases as well as other aggregate functions is in [5]. There we also confine things to one aggregate function per HAVING clause. This is not a problem in reality, as the more general case does not occur in the SkyServer query log at all.

4.4 Nested Queries

Queries of this type manifest themselves either explicitly with operators such as EXISTS, IN, ANY, ALL, or implicitly in some nested predicate, e.g., *T.u = (SELECT S.u FROM S WHERE S.v = 12)*. As with aggregate queries, we do not discuss every possible aspect of nested queries and refer to [5] for further information. Instead, we focus on several issues, and confine the presentation here to the EXISTS operator. To keep the exposition simple, we discuss nested queries of the following form, which covers all nested queries appearing in the log:

```
SELECT *
FROM [...]
WHERE [...]
OPT EXISTS(q_1)
...
OPT EXISTS(q_m)
```

where OPT is either AND or OR, and $\mathbf{q}_i$ is a query of the intermediate format. Further, each $\mathbf{q}_i$ refers to one single relation, and this relation does not appear in the FROM clause of the parent query. This also avoids any implicit self-join.

Given a nested query $\mathbf{q}$ of the above format, we transform it into the intermediate format as follows:

- Group m subqueries in the EXISTS clauses based on the relations they refer to. W.l.o.g., let the groups be $G_1 = \{\mathbf{q}_1^1, \ldots, \mathbf{q}_{m_1}^1\}, \ldots, G_l = \{\mathbf{q}_1^l, \ldots, \mathbf{q}_{m_l}^l\}$, where $l \leq m$, $\mathbf{q}_v^u \in \{\mathbf{q}_1, \ldots, \mathbf{q}_m\}$, and $\sum_{u=1}^{l} m_u = m$.
- Let $\mathbf{q}_i.FROM$ be the relation in the FROM clause of $\mathbf{q}_i$. Further, we write $\mathbf{q}_i.WHERE$ for all predicates together with their connections in the WHERE clause of $\mathbf{q}_i$. We transform $\mathbf{q}$ to the following query:

  ```
  SELECT *
  FROM [...], q_1.FROM, ..., q_m.FROM
  WHERE [...]
  OPT (q_1^1.WHERE OR ... OR q_{m_1}^1.WHERE)
  ...
  OPT (q_1^l.WHERE OR ... OR q_{m_l}^1.WHERE)
  ```

- Subsequent simple transformations may be required to convert the constraint in the WHERE clause of the above query to a conjunctive normal form.

We present three categories of nested queries having the above format to show why our transformation preserves the access area of $\mathbf{q}$ exactly. In fact, all nested queries with the EXISTS operator occurring in the SkyServer query log fall into these three categories.

LEMMA 4. *Consider the following query:*

```
SELECT *
FROM T
WHERE T.u > α
AND EXISTS
(SELECT * FROM S
WHERE S.u = T.u AND S.v < β)
```

where α and β are constants. The access area of this query is: $\sigma_{T.u>\alpha \wedge S.u=T.u \wedge S.v<\beta}(T \times S)$.

PROOF. The access area of this query is a subset of $T \times S$. We prove that an arbitrary element $(t, s) \in T \times S$ influences the result of the query if and only if $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta$.

($\Rightarrow$): Consider $(t, s) \in T \times S$ that influences the result. Then we have that $t.u > \alpha$. Next, if $s.v \geq \beta$, we can always remove (t, s) without influencing the result. So it must hold that $s.v < \beta$. Further, if there does not exist any $t' \in T$ such that $t'.u > \alpha$ and $s.u = t'.u$, then again we can safely remove (t, s). Thus, such a t' must exist. If $s.u \neq t.u$, as long as we keep (t', s), we can safely remove (t, s) while the result is not impacted. Hence, it holds that $s.u = t.u$. Thus, we have $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta$.

($\Leftarrow$): Let (t, s) be such that $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta$. We construct a database state where T contains only t, and S

contains only s. Clearly, if we remove (t, s), the result of the query in this database state is changed.

Combining $(\Rightarrow)$ and $(\Leftarrow)$, we conclude our proof. Using our procedure, we have $m = 1$, $\mathbf{q}_1.FROM$ is S, and $\mathbf{q}_1.WHERE$ is *S.u = T.u AND S.v* $< \beta$. Thus, the transformed query is:

```
SELECT *
FROM T, S
WHERE T.u > α AND S.u = T.u AND S.v < β
```

which is as expected. □

LEMMA 5. *Consider the following query:*

```
SELECT *
FROM T
WHERE T.u > α
AND EXISTS
(SELECT * FROM S
WHERE S.v < β AND S.u = T.u)
AND EXISTS
(SELECT * FROM S
WHERE S.v >= γ AND S.u = T.u)
```

where α, β, and γ are constants, and $\gamma \geq \beta$. The access area of this query is: $\sigma_{T.u>\alpha \wedge S.u=T.u \wedge (S.v<\beta \vee S.v\geq\gamma)}(T \times S)$.

PROOF. We prove that $(t, s) \in T \times S$ influences the result of the query if and only if: $t.u > \alpha \wedge s.u = t.u \wedge (s.v < \beta \vee s.v \geq \gamma)$.

$(\Rightarrow)$: If (t, s) influences the result, we have $t.u > \alpha$. If $\beta \leq s.v < \gamma$, we can safely discard (t, s) without influencing the result. Hence, it must hold that $s.v < \beta \vee s.v \geq \gamma$. By reasoning similarly to the proof of Lemma 4, we have $s.u = t.u$. Thus, if (t, s) influences the result, then: $t.u > \alpha \wedge s.u = t.u \wedge (s.v < \beta \vee s.v \geq \gamma)$.

$(\Leftarrow)$: Let (t, s) be such that $t.u > \alpha$, $s.u = t.u$, and $s.v < \beta \vee s.v \geq \gamma$. W.l.o.g., we assume that $s.v < \beta$. We construct a database state where T contains t only, and S contains s, and another s', where $s'.u = t.u$ and $s'.u \geq \gamma$. Then, if we remove (t, s), the query result in this database state is changed.

Using our procedure, we transform the original query to:

```
SELECT *
FROM T, S
WHERE T.u > α AND S.u = T.u
AND (S.v < β OR S.v >= γ)
```

which preserves the access area exactly. □

LEMMA 6. *Consider the following query:*

```
SELECT *
FROM T
WHERE T.u > α
OR EXISTS
(SELECT * FROM S
WHERE S.v < β AND S.u = T.u)
OR EXISTS
(SELECT * FROM S
WHERE S.v >= γ AND S.u = T.u)
```

where α, β, and γ are constants, and $\gamma \geq \beta$. The access area of this query is: $\sigma_{(T.u>\alpha \vee S.u=T.u) \wedge (T.u>\alpha \vee S.v<\beta \vee S.v\geq\gamma)}(T \times S)$.

PROOF. Following a proof similar to the one of Lemma 6, we derive that $(t, s) \in T \times S$ influences the result of the query if and only if $t.u > \alpha$, or $s.u = t.u \wedge (s.v < \beta \vee s.v \geq \gamma)$. Converting this Boolean expression to a conjunctive normal form, we arrive at the result. □

We can also use the three categories of nested queries discussed so far to extract access areas of nested queries that have more than one nested level, i.e., we are able to generalize beyond the query log of SkyServer. The following example illustrates our point.

EXAMPLE 4. *Consider the following query:*

```
SELECT *
FROM T
WHERE T.u > α
AND EXISTS
(SELECT * FROM S
WHERE S.u = T.u AND S.v < β
AND EXISTS
(SELECT * FROM R
WHERE R.v = S.v AND R.x < γ))
```

where α, β, and γ are constants. From Lemma 4, we know that in term of access area, the subquery of the outer EXISTS *operator is equivalent to:*

```
SELECT *
FROM S, R
WHERE S.u = T.u AND S.v < β
AND R.v = S.v AND R.x < γ
```

Here, we temporarily consider $T.u$ as a constant. With this transformation, the original query now has only one nested level. Continuing to apply Lemma 4, we transform the original query to:

```
SELECT *
FROM T, S, R
WHERE T.u > α AND S.u = T.u AND S.v < β
AND R.v = S.v AND R.x < γ
```

In addition, we can also process nested queries with aggregate subqueries by combining the theories of this section and of Section 4.3. Furthermore, we have an approximation scheme to process complex nested queries in general, i.e., the ones that do not conform to any format already discussed. The details of this approximation scheme and our handling of nested queries with other operators are in [5].

4.5 System Implementation

We now briefly describe our actual implementation. For a given query, we first parse it to identify its single fragments. We use JSqlParser[1], as it is an open source project under the LGPL license and has a powerful, extensible grammar that supports most of the SQL structure occurring in the SkyServer logs. Second, we transform these fragments into a form that conforms to our intermediate format (see Section 2.4). In particular, we extract the relations the query addresses, including any relation in any nested query, and the constraints on these relations and related columns. Third, we convert the constraints derived into conjunctive normal form. Finally, as a cleanup step, we replace any remaining alias with the real name of the relation and order the list of relations alphabetically. Besides this, we perform some consolidation on the remaining predicates: We remove redundant constraints, merge overlapping constraints, and check the set of constraints for contradictions.

5. OUR DISTANCE FUNCTION

Our end goal is to discover interesting access areas in the data space that may represent the user interests. To accomplish this, we

[1] `http://jsqlparser.sourceforge.net/home.php`

need to extract a bigger picture out of the access areas of similar queries. In other words, we need a procedure to aggregate the access areas of a large set of queries. We aim at achieving this by clustering queries based on overlap as our main objective of similarity. The distance measure that we use for clustering simply quantifies the overlap (i.e., the similarity of access areas). Please note that other distance measures could be used for this purpose. The distance does not even have to be a metric [16]. However, it should have its main focus on the content of queries and not their structure like in other cases [4]. As part of our aggregation, we define such a distance function as follows.

Consider two queries $\mathbf{q}_1$ and $\mathbf{q}_2$ of intermediate form (see Section 2.4). We define their distance as follows:

$$d(\mathbf{q}_1, \mathbf{q}_2) = d_{tables}(\mathbf{q}_1.FROM, \mathbf{q}_2.FROM) + d_{conj}(\mathbf{q}_1.WHERE, \mathbf{q}_2.WHERE) \quad (1)$$

where $\mathbf{q}.FROM$ denotes the tables in the access area of query $\mathbf{q}$, and $\mathbf{q}.WHERE$ denotes its WHERE part, which is its access area (in a conjunctive normal form). Note that with proper instantiation of d_{tables} and d_{conj}, one could actually compute this distance function on the raw queries as in [4]. However, we have shown that properly extracting access areas of queries is far from simply using as-is all predicates appearing in the queries. Instead, one needs to resort to our transformation of queries to intermediate format. In the followings, we provide our instantiation of d_{tables} and d_{conj}.

5.1 Distance of Access Tables d_{tables}

We use the Jaccard coefficient to measure the distance between the two sets of table names $\mathbf{q}_1.FROM$ and $\mathbf{q}_2.FROM$:

$$d_{tables}(\mathbf{q}_1.FROM, \mathbf{q}_2.FROM) = 1 - \frac{|\mathbf{q}_1.FROM \cap \mathbf{q}_2.FROM|}{|\mathbf{q}_1.FROM \cup \mathbf{q}_2.FROM|} \quad .$$

The Jaccard coefficient has the disadvantage that corner cases have to be defined if both queries do not access any table. This may occur if a query only queries database constants. In this case, we set d_{tables} to 0.

5.2 Distance of Conjunctions d_{conj}

Consider two Boolean expressions b_1 and b_2 which are both in conjunctive normal form. We define their distance to be:

$$d_{conj}(b_1, b_2) = \frac{\sum_{o_1 \in b_1} \min_{o_2 \in b_2} d_{disj}(o_1, o_2) + \sum_{o_2 \in b_2} \min_{o_1 \in b_1} d_{disj}(o_1, o_2)}{|b_1| + |b_2|}$$

where each $o_1 \in b_1$ is a disjunction of Boolean expression(s), and $|b_1|$ is the number of disjunctions of b_1. We define each $o_2 \in b_2$ and $|b_2|$ similarly. In addition, $d_{disj}(o_1, o_2)$ is the distance between o_1 and o_2, which is given by:

$$d_{disj}(o_1, o_2) = \frac{\sum_{p_1 \in o_1} \min_{p_2 \in o_2} d_{pred}(p_1, p_2) + \sum_{p_2 \in o_2} \min_{p_1 \in o_1} d_{pred}(p_1, p_2)}{|o_1| + |o_2|}$$

where $p_1 \in o_1$ is an atomic predicate, and $|o_1|$ is the number of atomic predicates of o_1. We define each $p_2 \in o_2$ and $|o_2|$ similarly. The distance between two atomic predicates p_1 and p_2, given by $d_{pred}(p_1, p_2)$, is as follows:

p_1 and p_2 refer to the same single column. This means that they are column-constant atomic predicates (see Section 2.1). We refer to the column as a.

First, we assume that a is numerical. Let $MBR(a)$ be the minimum bounding box of the area of $dom(a)$ that are accessed by all queries in the log, including those having accessed the empty area of $dom(a)$. Further, let $access(a) = content(a) \cup MBR(a)$. Since a typically has a data type, $dom(a)$ and hence $access(a)$ are intervals with finite bounds. We set $d_{pred}(p_1, p_2) = \frac{\textit{overlap of intervals}}{\textit{width of } access(a)}$, i.e., the normalized overlap of two respective intervals. For instance, assume that p_1 is $a < 3$, p_2 is $a > 2$, and $access(a_1) = [0, 5]$. We have $d_{pred}(p_1, p_2) = 1/5 = 0.2$. Here we use $access(a)$ instead of $content(a)$ for normalization to cope with queries accessing the empty area.

On the other hand, if a is categorical, we denote $access(a)$ as the union between $content(a)$ and the set of values of $dom(a)$ accessed by all queries in the log. Further, we replace the overlap of intervals by the number of items p_1 and p_2 have in common, and the width of $access(a)$ by its cardinality. Note again that since a typically has a data type, $dom(a)$ and hence $access(a)$ have finite numbers of values.

p_1 and p_2 refer to different columns. We set $d_{pred}(p_1, p_2)$ to the proportion of the joint space of the involved columns occupied by p_1 and p_2. For instance, assume that p_1 is $a_1 < 3$, p_2 is $a_2 > 2$, and $access(a_1) = access(a_2) = [0, 5]$. We have $d_{pred}(p_1, p_2) = (3 \times 3)/(5 \times 5) = 0.36$.

5.3 Implementation Issues

To use our distance function, for each column a, we need to know $access(a)$. Since $content(a) \subset access(a)$, we need to first identify $content(a)$. Normally, this can be done by simply querying the database. However, when doing this, we got the timeout error for many columns, especially those belonging to large relations. The issue can be resolved in two ways: (1) interact with the domain experts, or (2) estimate $content(a)$ and hence $access(a)$ from the database content. The advantage of (2) is that the system implementation does not need to be configured by hand/modified when turning to another database. On the other hand, the results with (1) might be better. However, our concern with this study is to find out whether our approach as a whole is practical, and whether results already are useful with relatively simple technical means. Optimizing output quality further is future work. This is why we have resorted to (2) in this current evaluation, as follows.

For each numerical column, we derive its statistics by querying a sample of its data, e.g., 100 rows, from SkyServer. Assume that $[m, M]$ is its value range obtained from the sample. Then, we set $access(a) = content(a) = \left[m - \frac{M-m}{2}, M + \frac{M-m}{2}\right]$, i.e., we double the size of the sampled range. When processing each query in the log containing a column-constant predicate of the form "a θ c", if it accesses data not falling into $access(a)$, we update this range accordingly.

For each categorical column, we do similarly. However, instead of the range, we maintain its set of values. If a query accesses a value that does not appear in this set, we update the set accordingly.

We stress again that our access to the SkyServer database at this point is only for this specific clustering purpose, and our extraction of access areas is not involved in any physical database access.

6. CASE STUDY: CLUSTERING TRANSFORMED DATA WITH DBSCAN

In this section, we present a case study where we cluster the transformed data using our distance function. Our objective is to find out if we can discover interesting aggregate access areas. Regarding clustering, there is a variety of existing algorithms in the literature. On the other hand, we want to find out whether results generated with relatively simple means are helpful from the perspective of a domain expert. Hence, we use a well-known, relatively simple, noise-aware algorithm, that does not need us to specify the number of clusters, namely DBSCAN [10].

6.1 The Data

The log originally contains $12,442,989$ queries. We were able to extract the access area of $12,375,426$ queries, which is more than 99.4%, leaving $67,563$ queries without extraction. The leftover queries are in fact not accepted by the grammar of the JSqlParser. This is because they (a) contain errors, (b) use user-defined SkyServer-specific functions, or (c) are not SELECT queries but statements with CREATE TABLE or DECLARE (issued by SkyServer administrators). So except for the pathological queries, our method performs well in extracting access areas.

6.2 Access Areas

In preliminary experiments, we have observed that DBSCAN (or, at least, its implementation used here) has severe performance problems when applied to the entire set of transformed queries. Thus, the following results are obtained on a (not necessarily representative) sample consisting of $5,611,087$ access areas of queries. Each access area in this sample is constrained to contain only predicates of the form either "$a\ \theta\ c$" (column-constant) or "$a_1\ \theta\ a_2$" (column-column). This is to increase interpretability of the results. Of course, while taking all queries into account might be more informative, using that subset does not contradict our objectives: We want to find hotspots of user interest, and we want to see how much overlap there is with the actual database content. We also want to learn how effective standard tools (e.g., an off-the-shelf clustering algorithm) in this context actually are.

In general, access areas of individual queries do not convey much information to the database owner, but a summary of this data for all queries is definitely interesting. To this end, using DBSCAN, we cluster the access areas in the sample described above. For each output cluster, we derive its minimum bounding hyper-rectangle, which we interpret as the aggregated access area of the queries involved. During this process, we leave out extreme range bounds by applying the 3-standard deviation rule. This is to ensure the robustness of the results. Overall, we obtain 403 clusters. We list 24 representative clusters in Table 1. We choose these clusters since we find them to contain few columns and hence, easy to interpret. For each cluster, we present the following information:

- Cluster ID.
- Cardinality: The number of access areas (queries) falling into the cluster.
- Area coverage: $\frac{v_{access}}{v_{content}}$ where v_{access} is the volume of the aggregated access area, and $v_{content}$ is the volume of the database content.
- Object coverage: $\frac{n_{access}}{n_{content}}$ where n_{access} is the number of objects falling into the aggregated access area, and $n_{content}$ is the number of objects of the database content.
- Access area: A Boolean expression describing the aggregated access area.

Going over the result, we find that most queries in each cluster are issued by different users, i.e., the cardinality of each cluster is approximately equal to the number of users. For each of the Clusters 1–17, its aggregated access area overlaps with a relatively small part of the database content. In particular, both of its area coverage and object coverage are fairly small (both less than 1% for Cluster 17). This shows that some users are interested in only a small part of the database content when issuing a query. We also see that while the area coverage is close to the object coverage in many clusters, this is not the case for Clusters 7, 8, 14, and 15. This is an indication that queries do not really follow the data distribution.

On the other hand, each of the Clusters 18–24 has its aggregated access area occupy empty area of the data space. Each such cluster contains from 17% (Cluster 20) up to 50% (Cluster 22) of the total number of queries accessing the same data space. This means that a significant number of queries refer to empty areas where no data objects are present.

6.3 Feedback from Domain Experts

This subsection describes qualitatitve feedback from our domain experts. This feedback is confined to the 24 representative clusters just described. Some important points are as follows:

The approach is promising. Both the SkyServer person and the independent astronomer have confirmed this. The results might not only be useful for the data owner, but for users as well: They help to explore the database, i.e., which combinations of attributes/attribute ranges obviously are important (e.g., Cluster 5). They also offer orientation in the sense 'Which parts of the data do others deem important?'.

Most results are plausible. Cluster boundaries tend to map to meaningful concepts of astronomy. For instance, the closer a dec-value is to the equator (dec-value 0), the more interesting the object is for an astronomer. The cluster in Figure 1(b) reflects this. However, not all cluster-boundary values are fully clear. Again in Figure 1(b), we do not know yet why the cluster boundary is $dec = 10$ (and not, say, 8 or 15, which would be just as conceivable). Similarly, we do not have an explanation for the specific boundary values along id-type attributes (Clusters 1–4 for instance). These clusters are numerous and require further investigation. On the other hand, our table does not contain clusters on attributes that the astronomer expects to be queried frequently, such as magnitude.

Result presentation should be improved/refined. Both our experts have interpreted the top row as the 'most frequent access area' (and were puzzled that most queries explicitly refer to *Photoz.objid*), but this is not correct in general. Other attributes may be queried more frequently, but the values in queries are spread more evenly over the range, i.e., there is no cluster. This is even likely, since the number of queries per cluster is relatively small ($179,072$ at best, compared to several million queries). What one can infer from the first row is that the values in that range are more likely to be referred to in queries than just outside of the range. A follow-up is that it would be interesting to know how much denser each cluster is, in contrast to its immediate surroundings. We conclude that we should have explained our results more extensively right away, and that there is further information of interest, such as that density drop.

Our results contain useful hints on how the database could be improved. To illustrate, *zooSpec.dec* is queried rather frequently with value -100, even though it is an angle and can only have -90 as its minimal value. Different steps are conceivable, e.g., a tighter definition of value ranges, or a better documentation.

There still are open questions which might be relevant for future research. For example, the astronomer has pointed out that there might be 'test queries' (i.e., queries that are exploratory in nature) which are numerous and influence the clustering result by much and 'final queries', as he calls it. While there might be only relatively few of them, they are important. Finding ways to differentiate between these categories, possibly based on the metadata available, is future work. There also are points that are minor in

Cluster	**Cardinality**	**Area Coverage**	**Object Coverage**	**Access area**
1	$179,072$	0.24	0.36	$1,237,657,855,534,432,934 \leq Photoz.objid \leq 1,237,666,210,342,830,434$
2	$121,311$	0.19	0.22	$1,115,887,524,498,139,136 \leq SpecObjAll.specobjid \leq 2,183,177,975,464,224,768$
3	$92,177$	0.22	0.21	$1,345,591,721,622,267,904 \leq galSpecLine.specobjid \leq 2,007,633,797,213,874,176$
4	$90,047$	0.25	0.25	$1,416,192,325,597,030,400 \leq galSpecInfo.specobjid \leq 2,183,213,984,470,034,432$
5	$90,015$	0.19	0.25	$PhotoObjAll.ra \leq 210 \wedge PhotoObjAll.dec \leq 10$
6	$82,196$	0.23	0.24	$1,228,357,946,564,438,016 \leq sppLines.specobjid \leq 2,069,493,422,263,134,208$
7	$23,021$	0.17	0.04	$54 \leq SpecObjAll.ra \leq 115$
8	$23,021$	0.23	0.09	$60 \leq SpecPhotoAll.ra \leq 124$
9	$18,904$	0.03	0.01	$(SpecObjAll.class = \text{'star'})$ $\wedge(51,578 \leq SpecObjAll.mjd \leq 52,178) \wedge (296 \leq SpecObjAll.plate \leq 3,200)$
10	$10,141$	0.26	0.27	$(DBObjects.access = \text{'U'}) \wedge ((DBObjects.type = \text{'V'}) \vee (DBObjects.type = \text{'U'}))$
11	$4,006$	0.24	0.18	$55 \leq emissionLinesPort.ra \leq 141$
12	$3,785$	0.21	0.17	$62 \leq stellarMassPCAWisc.ra \leq 138$
13	$1,622$	0.12	0.11	$AtlasOutline.objid > 1,237,676,243,900,255,188$
14	$1,371$	0.16	0.01	$(2 \leq zooSpec.ra \leq 120) \wedge (30 \leq zooSpec.dec \leq 70)$
15	$1,141$	0.10	0.05	$0 \leq Photoz.z \leq 0.1$
16	$1,102$	0.25	0.17	$(0 \leq galSpecExtra.bptclass \leq 3)$ $\wedge(galSpecExtra.specobjid = galSpecIndx.specObjID)$
17	$1,035$	< 0.001	< 0.001	$(sppLines.gwholemask = 0) \wedge (0 \leq sppLines.gwholeside \leq 50)$ $\wedge(sppLines.specobjid = sppParams.specobjid)$ $\wedge(-0.3 \leq sppParams.fehadop \leq 0.5) \wedge (2 \leq sppParams.loggadop \leq 3)$
18	$48,470$	0.0	0.0	$(10 \leq PhotoObjAll.ra \leq 120) \wedge (-90 \leq PhotoObjAll.dec \leq -50)$
19	$41,599$	0.0	0.0	$3,519,644,828,126,257,152 \leq galSpecLine.specobjid \leq 5,788,299,621,113,984,000$
20	$18,444$	0.0	0.0	$3,519,644,828,126,257,152 \leq galSpecInfo.specobjid \leq 5,788,299,621,113,984,000$
21	$18,043$	0.0	0.0	$4,037,480,726,273,651,712 \leq spplines.specobjid \leq 5,788,299,621,113,984,000$
22	$1,358$	0.0	0.0	$(6 \leq zooSpec.ra \leq 115) \wedge (-100 \leq zooSpec.dec \leq -15)$
23	422	0.0	0.0	$-0.98 \leq Photoz.z \leq -0.1$
24	217	0.0	0.0	$3.0 \leq Photoz.z \leq 6.5$

Table 1: Some aggregated access areas (clusters of queries), extracted from SkyServer query log.

nature, e.g., suggestions for further figures describing the clusters.

6.4 Comparison to [4]

We want to learn whether our distance measure affects the result by much. To do so, we compare our method to OLAPClus [4], which has a proprietary distance function (i.e., based on structure) for measuring the (dis-)similarity of (OLAP) queries. The distance function is also applicable to access areas. However, it requires *exact matching* of two atomic predicates and not their overlapping in access areas. Thus, when queries accessing the same data space have very different predicates, e.g., accessing different sets of objects, it is expected that OLAPClus does not group them into the same cluster, i.e., aggregated access areas are lost.

The result of OLAPClus on our data set of SkyServer access areas actually reflects this hypothesis. In particular, OLAPClus produces approximately $100,000$ clusters for Cluster 1 of our method. This is because almost every query in Cluster 1 has its predicate in the form $Photoz.objid = c$ where c is a constant. Similarly, for each of the Clusters 2–4 of our method, OLAPClus outputs about $50,000$ clusters. This does not only cause high redundancy but also loss of knowledge on the interests of users.

The benefits of our method on the other hand are two-fold: (a) succinct output that facilitates post-analysis, and (b) meaningful capture of the access patterns of users.

6.5 Comparison to OLAPClus on Raw Queries

Equation (1) suggests that one could actually obtain aggregated access areas by computing our distance function on the raw queries and clustering them accordingly, without using our extraction method. Essentially this can be done by applying OLAPClus. However, for fair comparison, we replace the exact matching of atomic predicates in OLAPClus by our d_{conj} (see Section 5).

The results show that this version of OLAPClus breaks Clusters 2, 5, 8, 9, 11, 12, 18, 19, 20, and 22 in Table 1. This is because these clusters contain queries of the forms in Section 4.3 and we have proved that directly using predicates as-is may lead to misleading access areas. In addition, the modified OLAPClus yields clusters that do not permit an easy construction of aggregated access areas, as heterogeneous Boolean expressions, resulted from keeping predicates as-is, are put in the same cluster.

6.6 Comparison to Re-querying

Next, we compare our method against the approach that re-issues queries for collecting statistics. Here, we use two performance metrics: efficiency (runtime) and quality of access areas.

Efficiency. Our method processes $100,000$ queries in about 45 seconds on our test machine (Intel® i5-750 CPU with 8GB RAM). There are however queries where the extraction takes rather long time, and in very rare cases, the extraction could not be done within reasonable time (in the range of hours). Looking closer, we find that query execution times of each single step (Parsing, Extraction, CNF, Consolidation) varies between: (a) Parsing: <1 millisecond and 94 milliseconds, (b) Extraction: <1 millisecond and 1333 milliseconds, (c) CNF: <1 millisecond and *undefined*, and (d) Consolidation: <1 millisecond and 95 milliseconds.

The CNF converter, which we took from an open source project, is definitely the weakest point with respect to efficiency. In particular, we discover that the necessary system resources (CPU time, RAM) grow exponentially with the number of predicates the access area currently processed includes. Fortunately, in our total of more than 12 million queries, there are only 471 queries with more than 35 predicates. Such a query often poses a performance bottleneck. To alleviate the issue, we provide a method within our implementation that only considers the first 35 predicates of any query. With this workaround, CNF conversion never lasts longer than 1 hour. More sophisticated processing is left for future work.

Re-issuing queries is far more expensive than our method. In particular, our method is orders of magnitude faster than this naïve approach. Since re-issuing a large number of queries against the SkyServer database does not terminate within a reasonable amount of time, we re-run the queries instead on a sample of the database. The result of this variant is discussed next.

Quality. Compared to re-issuing queries, our method of extracting access areas from the query log provides two advantages. First, we discover empty spaces that are accessed by many queries. As expected, extracting access areas from actual results of queries only yields the areas covered by the database content, i.e., Clusters 18–24 discovered by our method are missed by this approach. Second, our method is able to extract access areas from $1,220,358$ queries that cause errors when being issued to the SkyServer database. Further, our method even extracts access areas of queries that are not written in correct MSSQL code (which is necessary for SkyServer). Such queries often are written in a MySQL dialect such as *SELECT Galaxies.objid FROM Galaxies LIMIT 10*.

The approach that re-issues queries in turn neither yields empty spaces nor is able to process queries with execution errors. All in all, our method offers a more flexible solution towards extracting the access patterns of users from query logs.

7. CONCLUSIONS

Extracting access patterns of database users, i.e., access areas of queries, from query logs is crucial to learn the database usage. This has many applications; one is that it allows to make explicit the research focus of the respective scientific community. However, the task is challenging due to the lack of (a) a formal definition of access area, (b) a mapping of queries to their access areas, and (c) a procedure, including a distance function, to aggregate the access areas of a large set of queries.

In this paper, we have a proposed a solution to each of these challenges. First, we have introduced the novel concept of query access area. It allows the extraction of access areas independent of the database content. Second, we provide a mapping of our notion to all query types occurring in the log, i.e., we enable extraction of access areas in practice. Third, we exploit query overlap for the detection of aggregated access areas that abstract from the individual queries. Domain experts deem our approach interesting. Our case study on the SkyServer query log further shows that our method discovers clusters of access areas that occupy a small fraction of the database content. Some access areas even span empty parts of the data space. Empirical results also show that our method outperforms both a state of the art technique on measuring similarities of queries and an approach that re-issues queries.

In furture work, we plan to experiment with different clustering techniques on our data sets of extracted access areas. Further, we intend to test our method with different distance functions to unveil other interesting access patterns of SkyServer users.

Acknowledgment

We thank Jordan Raddick from Johns Hopkins University for helpful discussion and comments.

8. REFERENCES

[1] R. Agrawal et al. Context-sensitive ranking. In *SIGMOD Conference*, 2006.

[2] J. Akbarnejad et al. Sql QueRIE recommendations. *PVLDB*, 3(2), 2010.

[3] J. Aligon et al. Mining preferences from OLAP query logs for proactive personalization. In *ADBIS*, 2011.

[4] J. Aligon et al. Similarity measures for OLAP sessions. *Knowl. Inf. Syst.*, 37(2), 2014.

[5] F. Becker. Transforming the SDSS SkyServer SQL query log. Master's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2013.

[6] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Trans. Softw. Eng.*, 11(4), 1985.

[7] G. Chatzopoulou et al. Query recommendations for interactive database exploration. In *SSDBM*, 2009.

[8] G. Chatzopoulou et al. The QueRIE system for personalized query recommendations. *IEEE Data Eng. Bull.*, 34(2), 2011.

[9] A. Cleve and J.-L. Hainaut. Dynamic analysis of SQL statements for data-intensive applications reverse engineering. In *WCRE*, 2008.

[10] M. Ester et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[11] W. Gatterbauer. Databases will visualize queries too. *PVLDB*, 4(12), 2011.

[12] A. Ghosh et al. Plan selection based on query clustering. In *VLDB*, 2002.

[13] A. Giacometti et al. Recommending multidimensional queries. In *DaWaK*, 2009.

[14] T. Grust et al. True language-level SQL debugging. In *EDBT*, 2011.

[15] Y. Ioannidis. From databases to natural language: The unusual direction. In *NLDB*, 2008.

[16] A. K. Jain et al. Data clustering: A review. *ACM Comput. Surv.*, 31(3), 1999.

[17] K. P. Joshi et al. Warehousing and mining web logs. In *WIDM*, 1999.

[18] N. Koudas et al. Relaxing join and selection queries. In *VLDB*, 2006.

[19] G. Koutrika et al. Explaining structured queries in natural language. In *ICDE*, 2010.

[20] S. Mittal et al. QueRIE: A query recommender system supporting interactive database exploration. In *ICDMW*, 2010.

[21] H. Pirahesh et al. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD Conference*, 1992.

[22] F. Silvestri. Mining query logs: Turning search usage data into knowledge. *Found. Trends Inf. Retr.*, 4(1-2), 2010.

[23] V. Singh et al. SkyServer traffic report - the first five years. *CoRR*, abs/cs/0701173:190–204, 2007.

[24] X. Yang et al. Recommending join queries via query log analysis. In *ICDE*, 2009.

[25] Q. Yao et al. Finding and analyzing database user sessions. In *DASFAA*, 2005.

[26] J. Zhang. *Data use and access behavior in escience—exploring data practices in the new data-intensive science paradigm*. PhD thesis, Drexel University, Philadelphia, PA, USA, 2011.