Sieve: A Middleware Approach to Scalable Access Control for Database Management Systems

Primal Pappachan*, Roberto Yus*, Sharad Mehrotra*, Johann-Christoph Freytag**
*University of California, Irvine,**Humboldt-Universität zu Berlin
{primal, sharad}@ics.uci.edu, ryuspeir@uci.edu, freytag@informatik.hu.berlin.de

ABSTRACT

Current approaches for enforcing Fine Grained Access Control (FGAC) in DBMS do not scale to scenarios when the number of access control policies are in the order of thousands. This paper identifies such a use case in the context of emerging smart spaces wherein systems may be required by legislation, such as Europe's GDPR and California's CCPA, to empower users to specify who may have access to their data and for what purposes. We present Sieve, a layered approach of implementing FGAC in existing DBMSs, that exploits a variety of their features (e.g., UDFs, index usage hints, query explain) to scale to a large number of policies. Given a query, Sieve exploits its context to filter the policies that need to be checked. It also generates *guarded expressions* that save on evaluation cost by grouping policies and exploit database indices to cut on read cost. Our experimental results demonstrate that existing DBMSs can utilize Sieve to significantly reduce query-time policy evaluation cost. Using Sieve DBMSs can support real-time access control in applications such as emerging smart environments.

PVLDB Reference Format:
Primal Pappachan, Roberto Yus, Sharad Mehrotra, Johann-Christoph Freytag. Sieve: A Middleware Approach to Scalable Access Control for Database Management Systems. *PVLDB*, 13(11): 2424-2437, 2020.
DOI: https://doi.org/10.14778/3407790.3407835

1. INTRODUCTION

Organizations today capture and store large volumes of personal data that they use for a variety of purposes such as providing personalized services and advertisement. Continuous data capture, whether it be through sensors embedded in physical spaces to support location-based services (e.g., targeted ads and coupons), or in the form of web data (e.g., click-stream data) to learn users' web browsing habits, has significant privacy implications [6, 7, 33]. Regulations, such as the European General Data Protection Regulation (GDPR) [3], the California Online Privacy Protection Act (CalOPPA) [2], and the Consumer Privacy Act (CCPA) [1], have imposed legislative requirements that control how organizations manage user data. These requirements include data collection transparency, data minimization, and data retention. Another key requirement, for organizations/services to collect and to use an individual's data, is to adopt the principle of *choice and consent* [21][1]. This requirement has resulted in supporting mechanisms which allow users to opt-in/out and/or to specify data retention policies.

While such coarse level policies have sufficed for the web domain, recent work has argued that as smart spaces become pervasive wherein sensors continuously monitor individuals (e.g., continuous physiological monitoring by wearable devices, location monitoring both inside and outside buildings), systems will need to empower users with finer control over who can access their data and for what purpose. Supporting such fine grained policies raises several significant challenges that are beginning to attract research attention. These challenges include policy languages suitable for representing data capture, processing, sharing, and retention policies [28] together with mechanisms for users to specify their choices within the system. This paper addresses one such challenge: scaling enforcement of access control policies in the context of database query processing when the set of policies becomes a dominant factor/bottleneck in the computation due to their large number. This has been highlighted as one of the open challenges for Big Data management systems in recent surveys such as [17].

In the envisioned system that drives our research, data is dynamically captured from sensors and shared with people via queries based on user-specified access control policies. We describe a motivating use case of a smart campus in Section 2.1 which shows that data involved in processing a simple analytical query might require checking against hundreds to thousands of access control policies. Enforcing that many access control policies in real-time during query execution is well beyond database systems today. While our example and motivation is derived from the smart space and IoT setting, the need for such query processing with a large number of policies also applies to many other domains. This applicability will only increase as emerging legislation such as GDPR empowers users to control their data.

Today, database management systems (DBMSs) implement Fine-Grained Access Control (FGAC) by one of two mechanisms [8]: 1) *Policy as schema* and 2) *Policy as data*.

[1]Currently, such organizations typically follow the principle of *notice* wherein they inform the user about data collection, but may not support mechanisms to seek consent.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407835

In the former case, access control policies are expressed as authorization views [31]. Then, the DBMS rewrites the query and executes it against the relevant views instead of the original data. These views allow administrators to control access to a subset of the columns and rows of a table. In the latter case policies are stored in tables, just like data. The DBMS rewrites queries to include the policy predicates prior to execution [5, 9, 13, 15]. This mechanism allows administrators to express more fine-grained policies compared to views. Existing DBMS support both mechanisms, as they are both based on query rewriting [32], by appending policies as predicates to the `WHERE` clause of the original query. However, they are limited in the complexity of applications they can support due to the increased cost of query execution when the rewriting includes a large number of policies. Thus, scalable access control-driven query execution presents a novel challenge.

In this paper we propose Sieve, a general purpose middleware to support access control in DBMSs that enables them to scale query processing with large number of access control policies. It exploits a variety of features (index support, UDFs, hints) supported by modern DBMSs to scale to a large number of policies. A middleware implementation, layered on top of an existing DBMS, allows us to test Sieve independent of the specific DBMS used. This is particularly useful in our case (motivated by IoT) since different systems offer different trade-offs in IoT settings as highlighted in [18]. The comparative simplicity of implementing the technique in middleware enables us to explore the efficacy of different ideas instead of being constrained by the design choice of a specific system, as shown in previous work such as [14].

Sieve incorporates two distinct strategies to reduce overhead: reducing the number of tuples that have to be checked against complex policy expressions and reducing the number of policies that need to be checked against each tuple. First, given a set of policies, it uses them to generate a set of *guarded expressions* that are chosen carefully to exploit the best existing database indexes, thus reducing the number of tuples against which the complete and complex policy expression must be checked. This strategy is inspired by the technique for predicate simplification to exploit indices developed in [11]. Second, Sieve reduces the overhead of dynamically checking policies during query processing by filtering policies that must be checked for a given tuple by exploiting the context present in the tuple (e.g., user/owner associated with the tuple) and the query metadata (e.g., the person posing the query –i.e., querier– or their purpose). We define a policy evaluation operator Δ for this task and present an implementation as a User Defined Function (UDF).

Sieve combines the above two strategies in a single framework to reduce the overhead of policy checking during query execution. Thus, Sieve adaptively chooses the best strategy possible given the specific query and policies defined for that querier based on a cost model estimation. We evaluate the performance of Sieve using a real WiFi connectivity dataset captured in our building at UC Irvine, including connectivity patterns of over 40K unique devices/individuals. On this real dataset, we generate a synthetic set of policies that such individuals could have defined to control access to their data by others. We also test the performance of our system on a synthetic dataset based on a smart mall where connectivity data of devices are logged inside shops in the mall. Our results highlight the benefit of Sieve-generated query rewrite when compared to the traditional query rewrite approach for access control when processing different queries. Additionally, we perform these experiments on two different DBMSs, MySQL and PostgreSQL, showcasing Sieve's abilities as a middleware.

Outline of the paper. Section 2 presents a case study of a real IoT deployment, with a large set of access control policies defined, and reviews related work. Section 3 formalizes the query and policy model, and the access control semantics used by Sieve. We also describe an overview of the approach used by Sieve with an outline of two different strategies. Section 4 presents an algorithmic solution for the first strategy i.e., to generate appropriate *guarded expressions*. Section 5 describes the details of the Sieve generated query rewrite along with various optimization techniques used. Section 6 presents the experimental evaluation using two different datasets and two different DBMSs. Finally, Section 7 presents conclusions and future works.

Extended version. Due to space limitations, we could not describe several technical details, which can be found in the extended version [30], including: dynamic generation of guarded expressions when the policy dataset is not static, persistence of policies and guards, full implementation of policy evaluation operator, experimental details (relation schemas, sample policies), and more experimental results.

2. PROBLEM SETTING

We present a case study based on a smart campus setting where there are a large number of FGAC policies specified by users for their collected data. Using this context, we review access control strategies in the literature and show that they fall short when enforcing large policy sets.

2.1 Smart Campus Case Study

We consider a motivating application wherein an academic campus supports variety of smart data services such as real-time queue size monitoring in different food courts, occupancy analysis to understand building usage (e.g., room occupancy as a function of time and events, determining how space organization impacts interactions amongst occupants, etc.), or automating class attendance and understanding correlations between attendance and grades [20]. While such solutions present interesting benefits, such as improving student performance [20] and better space utilization, there are privacy challenges [29] in the management of such data. This case study is based on our own experience building a smart campus with variety of applications ranging from real-time services to offline analysis over the past 4 years. The deployed system, entitled TIPPERS [26], is in daily use in several buildings in our UC Irvine campus[2]. TIPPERS at our campus captures connectivity events (i.e., logs of the connection of devices to WiFi APs) that can be used, among other purposes, to analyze the location of individuals to provide them with services.

We use the UC Irvine campus, with the various entities and relationships presented in Figure 1 (along with the expected number of members in brackets), as a use case. Consider a professor in the campus posing the following analytical query to evaluate the correlation between regular atten-

[2]More information about the system and the applications supported at `https://tippers.ics.uci.edu`

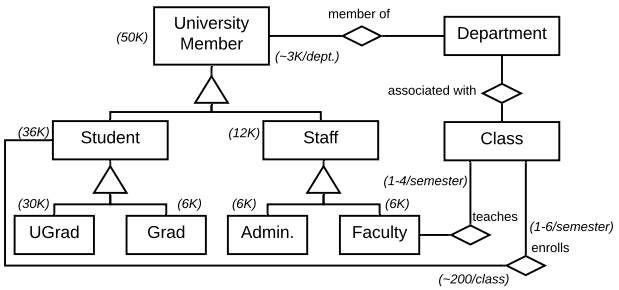

Figure 1: Entities and relationships in a Smart Campus Scenario.

dance in her class vs. student performance at the end of the semester:

```
StudentPerf(WifiDataset, Enrollment, Grades)=
(SELECT student, grade, sum(attended)
 FROM (
  SELECT W.owner AS student, W.ts-date AS date,
       count(*)/count(*) AS attended
  FROM WiFiDataset AS W, Enrollment AS E
  WHERE E.class="CS101" AND E.student=W.owner AND W.
       ts-time between "9am" AND "10am" AND W.ts-date
        between "9/25/19" AND "12/12/19" AND W.wifiAP
       ="1200"
  GROUP BY W.owner, W.ts-date) AS T, Grades AS G
 WHERE T.student=G.student
 GROUP BY T.student)
```

Let us assume that within the students in the professor's class, there exist different privacy profiles (as studied in the mobile world by Lin et al. [24]). Adapting the distribution of users by profile to our domain, we assume that 20% of the students might have a common default policy ("unconcerned" group), 18% may want to define their own precise policies ("advance users"), and the rest will depend on the situation (for which we consider, conservatively, 2/3 to be "unconcerned" and 1/3 "advance"). Using this distribution of privacy profiles and applying it to a class of 200 students, we have 120 unconcerned users who will adopt the default policy and 80 advanced users who will define their own set of policies. With the conservative assumption that there are two default policies per default user and at least 4 specific policies per advanced user, we have a total of 560 policies. Typically, advanced users define more policies than this conservative assumption so if we were to add two additional policies per group it will increase the number of policies to 880, or 1.2K (with three additional policies per group).

Given these numbers, together with students taking 1-6 classes per semester, the above query executed over classes a professor taught over the year (faculty teaches 1-4 classes per semester) would involve 3.3K (560 policies/class * 2 classes/quarter * 3 quarters/year) to 7.2K (using 1.2K policies/-class estimation) policies. We only focused on a single data type captured in this analysis (i.e., connectivity data) with two conditions per policy (e.g., time and location) and policies defined by a given user at the group-level (and not at the individual-level, which will even further increase the number of policies).

2.2 Related Work

As discussed in the comprehensive survey of access control in databases in [8], techniques to support FGAC can be broadly classified as based on views (e.g.,authorization views [31] and Oracle Virtual Private Database [25]) or based on storing policies in the form of data (e.g., Hippocratic databases [5] and the follow up work [23, 4]). In either of these approaches, input queries are rewritten to filter out tuples for which the querier does not have access permission. The view-based approach would be infeasible given the potentially large number of queriers/purposes which would result in creating and maintaining materialized views for each of them. In the policy-as-data based approach, the enforcement results on computationally expensive query processing. This is because the rewrite is done by adding conditions to the query's `WHERE` clause as ⟨`query predicate`⟩ `AND (`P_1 `OR ... OR` P_n`)` (where each P_i above refers to the set of predicates in each policy) or by using case-statement and outer join. In a situation like the one in our use case study, it results in appending hundreds of policy conditions to the query in a disjunctive normal form which adds significant overheads. Both strategies currently do not scale to scenarios with large number of policies.

Other approaches, such as [10], have proposed augmenting tuples with the purpose for which they can be accessed. This reduces the overheads at query time and as policy checking could be performed at data ingestion. Such pre-processing based approaches have significant limitations in the context where there are large number of fine-grained polices such as in the context that motivates our work. Determining permissions for individuals and encoding them as columns or multiple rows can result in exorbitant overhead during ingestion, specially when data rates are high (e.g., hundreds of sensor observations per second). Additionally, pre-processing efforts might be wasted for those tuples that are not queried frequently or at all. Other limitations include: 1) Impossibility of pre-processing policy predicates that depend on query context or information that is not known at that time of insertion; and 2) Difficulty to deal with dynamic policies which can be updated/revoked/inserted at any time (thus requiring processing tuples already inserted when policies change). Recent work [12, 14], that performs some pre-processing for access control enforcement, limits pre-processing to policies explicitly defined to restrict user's access to certain type of queries or to certain tables. The checking/enforcement of FGAC at tuple level is deferred to query-time and enforced through query rewriting as is the case in our paper.

Several research efforts have focused on implementing access control in the context of the IoT and smart spaces. In [27], the authors propose an approach for policy evaluation on streaming sensor data punctuated with access control policies. Their approach does not handle analytical queries with policies on the arriving data. Additionally the implementation of their approach requires significant modification to existing DBMS to make different operators *security-aware* for a large number of policies. In [16], the authors proposed a new architecture based on MQTT for IoT ecosystems. However, like [27] the focus of this work is not on managing large number of policies at run time and hence, they would experience the same issues highlighted for traditional query rewrite strategies.

3. SIEVE APPROACH TO FGAC

We describe our modeling of the fundamental entities in policy-driven data processing: data, query, and policies. Using these, we describe the access control semantics used in this paper. We finish the section with a sketch of the approach followed by Sieve to speed up policy enforcement.

Table 1: Frequently used notations.

Notation	*Definition*
$\mathcal{D}$	Database
$i_i \in \mathcal{I}$	Index and set of indexes in $\mathcal{D}$
$r_i \in \mathcal{R}$	Relation and set of relations in $\mathcal{D}$
$u_k \in \mathcal{U}$	User and set of users in $\mathcal{D}$
$t_j \in \mathcal{T}$; $\mathcal{T}_{r_i}$; $\mathcal{T}_{Q_i}$; $\mathcal{T}_{p_l}$	Tuple and set of tuples: in $\mathcal{D}$; required to compute Q_i; controlled by p_l
group(u_k)	Groups u_k is part of
Q_i: $\mathtt{QM}^i$	Query; Metadata of Q_i
$p_l \in \mathcal{P}$; $\mathcal{P}_{\mathtt{QM}^i}$	Access control policy and set of policies in $\mathcal{D}$; set of policies related to a query given its metadata
$\mathtt{oc}_i^l \in \mathtt{OC}^l$;$\mathtt{qc}_i^l \in \mathtt{QC}^l$;$\mathtt{AC}^l$	Object conditions; querier conditions; action of p_l
$\mathcal{E}(\mathcal{P}) = \mathtt{OC}^1 \vee \cdots \vee \mathtt{OC}^{\lvert\mathcal{P}\rvert}$	*Policy expression* of $\mathcal{P}$
$\mathcal{G}(\mathcal{P}) = G_1 \vee \cdots \vee G_n$	*Guarded policy expression* of $\mathcal{P}$ (DNF of guarded expressions)
$G_i = \mathtt{oc}_g^i \wedge \mathcal{P}_{G_i}$	*Guarded expression* consisting of *guard* ($\mathtt{oc}_g^i$) and its *policy partition* ($\mathcal{P}_{G_i}$)
$\mathcal{CG}$	*Candidate guards* for $\mathcal{E}(\mathcal{P})$
$eval(exp, t_t)$	Function which evaluates a tuple t_t against a expression *exp*
$\Delta(G_i, \mathtt{QM}^i, t_t)$	Ppolicy evaluation operator
$\rho(pred)$	Cardinality of a predicate
c_e	Cost of evaluating a tuple against the set policies
c_r	Cost of reading a tuple using an index

We have summarized frequently used notations in Table 1 for perusal.

3.1 Modeling Policy Driven Data Processing

Data Model. Let us consider a database $\mathcal{D}$ consisting of a set of relations $\mathcal{R}$, a set of data tuples $\mathcal{T}$, a set of indexes $\mathcal{I}$, and set of users $\mathcal{U}$. $\mathcal{T}_{r_i}$ represents the set of tuples in the relation $r_i \in \mathcal{R}$. Users are organized in collections or *groups*, which are hierarchical (i.e., a group can be subsumed by another). For example, the group of undergraduate students is subsumed by the group of students. Each user might belong to multiple groups and we define the method *group*(u_k) which returns the set of groups u_k is member of. Each data tuple $t_j \in \mathcal{T}$ belongs to a $u_k \in \mathcal{U}$ or a group whose access control policies restrict/grant access over that tuple to other users. We assume that for each data tuple $t_j \in \mathcal{T}$ there exists an owner $u_k \in \mathcal{U}$ who owns it, whose access control policies restrict/grant access over that tuple to other users (the ownership can be also shared by users within a group).

Query Model. The SELECT-FROM-WHERE query posed by a user u_k is denoted by Q_i and tuples in the relations in the FROM statement(s) of the query are denoted by $\mathcal{T}_{Q_i} = \bigcup_{i=1}^{n} \mathcal{T}_{r_i}$. In our model, we consider that queries have associated metadata $\mathtt{QM}^i$ which consists of information about the querier and the context of the query. This way, we assume that for any given query Q_i, $\mathtt{QM}^i$ contains the identity of the querier (i.e., $\mathtt{QM}^i_{querier}$) as well as the purpose of the query (i.e., $\mathtt{QM}^i_{purpose}$). In the example query in Section 2.1, $\mathtt{QM}^i_{querier}$="Prof.Smith" and $\mathtt{QM}^i_{purpose}$="Analytics".

Access Control Policy Model. A user specifies an access control policy (in the rest of the paper we will refer to it simply as policy) to allow or to restrict access to certain data she owns, to certain users/groups under certain conditions. Let $\mathcal{P}$ be the set of policies defined over $\mathcal{D}$ such that $p_l \in \mathcal{P}$ is defined by a user u_k to control access to a set of data tuples in r_i. Let that set of tuples be $\mathcal{T}_{p_l}$ such that $\mathcal{T}_{p_l} \subseteq \mathcal{T}_{u_k} \cap \mathcal{T}_{r_i}$. We model such policy as $p_l = \langle \mathtt{OC}^l, \mathtt{QC}^l, \mathtt{AC}^l \rangle$, where each element represents:

• Object Conditions ($\mathtt{OC}^l$) are defined using a conjunctive boolean expression $\mathtt{oc}_1^l \wedge \mathtt{oc}_2^l \wedge ... \wedge \mathtt{oc}_n^l$ which determines the access controlled data tuple(s). Each *object condition* ($\mathtt{oc}_c^l$) is a boolean expression $\langle attr, op, val \rangle$ where *attr* is an attribute (or column) of r_i, *op* is a comparison operator (i.e., $=, !=, <, >, \geq, \leq$, IN, NOT IN, ANY, ALL), and *val* can be either: (1) A constant or a range of constants or (2) A derived value(s) defined in terms of the expensive operator (e.g., a user defined function to perform face recognition) or query on $\mathcal{D}$ that will obtain such values when evaluated. In this paper, we focus on the object conditions with values as constants. To represent boolean expressions involving a range defined by two comparison operators (e.g., $4 \leq a < 20$) we use the notation $\langle attr, op1, val1, op2, val2 \rangle$ (e.g., $\langle a, \geq, 4, <, 20 \rangle$). As an example, $\mathtt{oc}^l_{owner}$ is an $\mathtt{oc}_c^l \in \mathtt{OC}^l$ such that $\mathtt{oc}_c^l = \langle r_i.owner, =, u_k \rangle$ or $\mathtt{oc}_c^l = \langle r_i.owner, =, group(u_k) \rangle$.

• Querier Conditions ($\mathtt{QC}^l$) identify the metadata attributes of the query to which the access control policy applies. $\mathtt{QC}^l$ is a conjunctive boolean expression $\mathtt{qc}_1^l \wedge \mathtt{qc}_2^l \wedge \cdots \wedge \mathtt{qc}_m^l$. Our model is inspired by the well studied Purpose Based Access Control (Pur-BAC) model [9] to define the querier conditions. Thus, we assume that each policy contains has at least two querier conditions such as $\mathtt{qc}^l_{querier} = \langle \mathtt{QM}^i_{querier}, =, u_k \rangle$ or $\mathtt{qc}^l_{querier} = \langle \mathtt{QM}^i_{querier}, =, group(u_k) \rangle$ (that defines either a user or group), and a $\mathtt{qc}^l_{purpose} = \langle \mathtt{QM}^i_{purpose}, =, purpose \rangle$ which models the intent/purpose of the querier (e.g., safety, commercial, social, convenience, specific applications on the scenario, or any [22]). Other pieces of querier context (such as the IP of the machine from where the querier posed the query, or the time of the day) can easily be added as querier conditions although in the rest of the paper we focus on the above mentioned querier conditions.

• Policy Action ($\mathtt{AC}^l$) defines the enforcement operation, or *action*, which must be applied on any tuple $t_j \in \mathcal{T}_{p_l}$. We consider the default action, in the absence of an explicit policy allowing access to data, to be *deny*. Such a model is standard in systems that collect/manage user data. Hence, explicit access control actions associated with policies in our context are limited to *allow*.

Based on this policy model, we show two sample policies in the context of the motivating scenario explained before. First, we describe a policy with object conditions containing a constant value. This policy is defined by John to regulate access to his connectivity data to Prof. Smith only if he is located in the classroom and for the purpose of class attendance as follows: ⟨[W.owner = John ∧ W.ts-time ≥ 09:00 ∧ W.ts-time ≤ 10:00 ∧ W.wifiAP = 1200], [Prof. Smith ∧ Attendance Control], allow⟩. Second, we describe the same policy with an object condition derived from a query to express that John wants to allow access to his location data only when he is with Prof. Smith. The object condition is updated as: [W.owner = John ∧ W.wifiAP = (SELECT W2. wifiAP FROM WifiDataset AS W2 WHERE W2.ts-time = W.ts-time AND W2.owner = "Prof.Smith")]. Finally, if a user expresses a policy with a deny action (e.g., to limit the scope/-coverage of an allow policy), we can translate it into the explicitly listed allow policies. For instance, given an allow policy, "allow John access to my location" and an overlap-

ping deny policy from the same user "deny everyone access to my location when in my office", we express both by replacing the original allow policy by "allow John access to my location when I am in locations other than my office". We therefore restrict our discussions to allow policies.

Access Control Semantics. We define access control as the task of deriving $\mathcal{T}'_{Q_i} \subseteq \mathcal{T}_{Q_i}$ which is the projection of $\mathcal{D}$ on which Q_i can be executed with respect to access control policies defined for its querier. Thus, $\forall\ t_t \in \mathcal{T}_{Q_i},\ t_t \in \mathcal{T}'_{Q_i} \Leftrightarrow eval(\mathcal{E}(\mathcal{P}), t_t) = True$. The function $eval(\mathcal{E}(\mathcal{P}), t_t)$ evaluates a tuple t_t against the policy expression $\mathcal{E}(\mathcal{P})$ that applies to Q_i as follows:

$$eval(\mathcal{E}(\mathcal{P}), t_t) = \begin{cases} True & \text{if } \exists \quad p_l \in \mathcal{P} \mid \text{eval}(\mathtt{OC}^l, t_t) = \text{True} \\ False & \text{otherwise} \end{cases}$$

where $eval(\mathtt{OC}^l, t_t)$ evaluates the tuple against the object conditions of p_l as follows:

$$eval(\mathtt{OC}^l, t_t) = \begin{cases} True & \text{if } \forall\ \mathtt{oc}^l_c \in \mathtt{OC}^l \mid t_t.\text{attr} = \mathtt{oc}^l_c.\text{attr} \implies \\ & \text{eval}(\mathtt{oc}^l_c.\text{op}, \mathtt{oc}^l_c.\text{val}, t_t.\text{val}) = \text{True} \\ False & \text{otherwise} \end{cases}$$

where $eval(\mathtt{oc}^l_c.op, \mathtt{oc}^l_c.val, t_t.val)$ compares the object condition value ($\mathtt{oc}^l_c.val$) to the tuple value ($t_t.val$) that matches the attribute of the object condition, using the object condition operator. If the latter is a derived value, the expensive operator/query is evaluated to obtain the value.

This access control semantics satisfies the sound and secure properties of the correctness criterion defined by [34]. If no policies are defined on t_t then the tuple is not included in $\mathcal{T}'_{Q_i}$ as our access control semantics is opt-out by default. Depending upon the query operations, evaluating policies after them is not guaranteed to produce correct results. This is trivially true in the case for aggregation or projection operations that remove certain attributes from a tuple. In queries with non-monotonic operations such as set difference, performing query operations before policy evaluation will result in inconsistent answers. Let $\mathcal{P}$ be the set of policies defined on r_k that control access to Q_i (a query with a set difference). $\mathcal{E}(\mathcal{P})$ is the Disjunctive Normal Form (DNF) expression of $\mathcal{P}$ such that $\mathcal{E}(\mathcal{P}) = \mathtt{OC}^1 \vee \cdots \vee \mathtt{OC}^{|\mathcal{P}|}$ where $\mathtt{OC}^l$ is conjunctive expression of object conditions from $p_l \in \mathcal{P}$. After appending $\mathcal{E}(\mathcal{P})$ to Q_i we obtain: `SELECT * FROM` r_j `MINUS SELECT * FROM` r_k `WHERE` $\mathcal{E}(\mathcal{P})$. Consider a tuple $t_k \in \mathcal{T}_{r_k}$ which has policy $p_l \in \mathcal{P}$ that denies Q_i access to t_k. If there exists a tuple $t_j \in \mathcal{T}_{r_j}$ such that $t_j = t_k$, then performing set difference operations before checking policies on r_k will result in a tuple set that does not include t_j. On the other hand, if policies for r_k are checked first, then $t_k \notin \mathcal{T}_{Q_i}$ and therefore t_j will be in the query result.

3.2 Overview of Sieve Approach

For a given query Q_i, the two main factors that affect the time taken to evaluate the set of policies for the set of tuples $\mathcal{T}_{Q_i}$ required to compute Q_i (i.e., $eval(\mathcal{E}(\mathcal{P}), t_t)\ \forall\ t_t \in \mathcal{T}_{Q_i}$) are the large number of complex policies and the number of tuples in $\mathcal{T}_{Q_i}$. The overhead of policy evaluation can thus be reduced by first eliminating tuples using low cost filters before checking the relevant ones against complex policies and second by minimizing the length of policy expression a tuple t_t needs to be checked against before deciding whether it can be included in the result of Q_i or not. These two fundamental building blocks form the basis for Sieve.

- **Reducing Number of Policies.** Not all policies in $\mathcal{P}$ are relevant to a specific query Q_i. We can first easily filter out those policies that are defined for different queriers/purposes given the query metadata $\mathtt{QM}^i$. For instance, when Prof. Smith poses a query for grading, only the policies defined for him and the faculty group for grading purpose are relevant out of all policies defined on campus. We denote the subset of policies which are relevant given the query metadata $\mathtt{QM}^i$ by $\mathcal{P}_{\mathtt{QM}^i} \subseteq \mathcal{P}$ where $p_l \in \mathcal{P}_{\mathtt{QM}^i}$ iff $\mathtt{QM}^i_{purpose} = \mathtt{qc}^l_{purpose} \wedge (\mathtt{QM}^i_{querier} = \mathtt{qc}^l_{querier} \vee \mathtt{qc}^l_{querier} \in group(\mathtt{QM}^i_{querier}))$. In addition, for a given tuple $t_t \in \mathcal{T}_{Q_i}$ we can further filter policies in $\mathcal{P}_{\mathtt{QM}^i}$ that we must check based on the values of attributes in t_t. For instance, we can further restrict the set of policies relevant for Prof. Smith's query by considering tuple attributes such as its owner (i.e., $t_t.owner$) and only retrieving the relevant policies based on the attribute value.
- **Reducing Number of Tuples.** Even if the number of policies to check are minimized, the resulting expression $\mathcal{E}(\mathcal{P})$ might still be computationally complex. To speed up processing of $\mathcal{E}(\mathcal{P})$ further, we derive low cost filters (object conditions) from it which can filter out tuples by exploiting existing indexes $\mathcal{I}$ over attributes in the database. We therefore rewrite the policy expression $\mathcal{E}(\mathcal{P}) = \mathtt{OC}^1 \vee \cdots \vee \mathtt{OC}^{|\mathcal{P}|}$ as a *guarded policy expression* $\mathcal{G}(\mathcal{P})$ which is a disjunction of *guarded expressions* $\mathcal{G}(\mathcal{P}) = G_1 \vee \cdots \vee G_n$. Each G_i consists of a *guard* $\mathtt{oc}^i_g$ and a *policy partition* $\mathcal{P}_{G_i}$ where $\mathcal{P}_{G_i} \subseteq \mathcal{P}$. Note that $\mathcal{P}_{G_i}$ partitions the set of policies, i.e., $\mathcal{P}_{G_i} \cap \mathcal{P}_{G_j} = \emptyset\ \forall\ G_i, G_j \in \mathcal{G}(\mathcal{P})$. Also, all policies in $\mathcal{P}$ are covered by one of the guarded expressions, i.e., $\forall\ p_i \in \mathcal{P}$ ($\exists\ G_i \in G$ such that $p_i \in \mathcal{P}_{G_i}$). We will represent the guarded expression $G_i = \mathtt{oc}^i_g \wedge \mathcal{P}_{G_i}$ where $\mathcal{P}_{G_i}$ is the set of policies but for simplicity of expression we will use it as an expression where there is a disjunction between policies.

  The *guard* term $\mathtt{oc}^i_g$ is an object condition that can support efficient filtering by exploiting an index. In particular, it satisfies the following properties:

  - $\mathtt{oc}^i_g$ is a simple predicate over an attribute (e.g., $ts - time > 9am$) and the attribute in $\mathtt{oc}^i_g$ has an index defined on it (i.e., $\mathtt{oc}^i_g.attr \in \mathcal{I}$).
  - The guard $\mathtt{oc}^i_g$ is a part of all the policies in the partition and can serve as a filter for them $\mathcal{P}_{G_i}$ (i.e., $\forall\ p_l \in \mathcal{P}_{G_i}\ \exists\ \mathtt{oc}^l_j \in \mathtt{OC}^l \mid \mathtt{oc}^l_j \implies \mathtt{oc}^i_g$).

As an example, consider the policy expression of all the policies defined by students to grant the professor access to their data in different situations. Let us consider that many of such policies grant access when the student is connected to the WiFi AP of the classroom. For instance, in addition to John's policy defined before, let us consider that Mary defines the policy ⟨`[W.owner = Mary ∧ ∧ W.wifiAP = 1200], [Prof. Smith ∧ Attendance Control], allow`⟩. This way, such predicate (i.e., wifiAP=1200) could be used as a guard that will group those policies, along with others that share that predicate, to create the following expression: `wifiAP =1200 AND ((owner=John AND ts-time between 9 AND 10am OR (owner=Mary) OR ...)`.

Sieve adaptively selects a query execution strategy when

a query is posed leveraging the above ideas. First, given Q_i, Sieve filters out policies based on $\mathtt{QM}^i$. Then, using the resulting set of policies, it replaces any relation $r_j \in Q_i$ by a projection that satisfies policies in $\mathcal{P}_{\mathtt{QM}^i}$ that are defined over r_j. It does so by using the guarded expression $\mathcal{G}(\mathcal{P}_{r_j})$ constructed as a query `SELECT * FROM` r_j `WHERE` $\mathcal{G}(\mathcal{P}_{r_j})$.

By using $\mathcal{G}(\mathcal{P}_{r_j})$ and its guards $\mathtt{oc}_g^i$, we can efficiently filter out a high number of tuples and only evaluate the relevant tuples against the more complex policy partitions $\mathcal{P}_{G_i}$. The generation of $\mathcal{G}(\mathcal{P}_{r_j})$ might take place offline if the policy dataset is deemed to undergo small number of changes over time. Otherwise, the generation can be done either when a change is made in the policy table or at query time for more dynamic scenarios (our algorithm is efficient enough for dynamic scenarios as we show in Section 6).

A tuple that satisfies the guard $\mathtt{oc}_g^i$ is then checked against $\mathcal{E}(\mathcal{P}_{G_i}) = \mathtt{OC}^1 \vee \cdots \vee \mathtt{OC}^{|\mathcal{P}_{G_i}|}$. This evaluation could be expensive depending upon the number of policies in $\mathcal{P}_{G_i}$. As it is a DNF expression, in the worst case (a tuple that does not satisfy any policy) will have to be evaluated against each $\mathtt{OC}^j \in \mathcal{P}_{G_i}$. We introduce a policy evaluation operator ($\Delta(G_i, \mathtt{QM}^i, t_t)$) which takes a guarded expression G_i, query metadata $\mathtt{QM}^i$, and each tuple t_t that satisfied $\mathtt{oc}_g^i$ and retrieves a subset of $\mathcal{P}_{G_i}$ (filtered using $\mathtt{QM}^i$ and t_t). Then, policy evaluation on the tuples that satisfy the guard is only performed on this subset of policies instead of $\mathcal{P}_{G_i}$. Sieve situationally selects based on each $G_i \in G$ whether to use the policy evaluation operator for evaluating $\mathcal{P}_{G_i}$ to minimize the execution cost. We explain the details of implementation of this operator and the selection strategy in Section 5.

Hence, the main challenges are: 1) Selecting appropriate guards and creating the guarded expression; 2) Dynamically rewriting query by evaluating different strategies and constructing a query that can be executed in an existing DBMS. We explain our algorithm to generate guarded expressions for a set of policies in Section 4. We later explain how Sieve can be implemented in existing DBMSs and how it selects an appropriate strategy depending on the query and the set of policies that apply to the query.

4. CREATING GUARDED EXPRESSIONS

Our goal is to translate a policy expression $\mathcal{E}(\mathcal{P}) = \mathtt{OC}^1 \vee \cdots \vee \mathtt{OC}^{|\mathcal{P}|}$ into a guarded policy expression $\mathcal{G}(\mathcal{P}) = G_1 \vee \cdots \vee G_n$ such that the cost of evaluating $\mathcal{G}(\mathcal{P})$ given database $\mathcal{D}$ and set of indices $\mathcal{I}$ is minimized

$$\min cost(\mathcal{G}(\mathcal{P})) = \min \sum_{G_i \in G} cost(G_i) \tag{1}$$

where G is the set of all the guarded expressions in $\mathcal{G}(\mathcal{P})$. A guarded expression G_i corresponds to $G_i = \mathtt{oc}_g^i \wedge \mathcal{P}_{G_i}$ where $\mathtt{oc}_g^i$ is a guard and $\mathcal{P}_{G_i}$ is a policy partition. The cost of evaluating a tuple against a set of policies is defined by

$$cost(eval(\mathcal{E}(\mathcal{P}_{G_i}))) = \alpha.|\mathcal{P}_{G_i}|.c_e \tag{2}$$

where α represents the average number of policies in $\mathcal{P}_{G_i}$ that the tuple t_t is checked against from the disjunctive expression in $\mathcal{E}(\mathcal{P}_{G_i})$ (we assume that the DBMS stops the execution of such a disjunctive expression with the first policy that the tuple satisfies and skips the rest), and c_e represents the average cost of evaluating t_t against the set of object conditions for a policy $p_l \in \mathcal{P}_{G_i}$ (i.e., $\mathtt{OC}^l$). We model $cost(G_i)$ as

$$cost(G_i) = \rho(\mathtt{oc}_g^i).(c_r + cost(eval(\mathcal{E}(\mathcal{P}_{G_i})))) \tag{3}$$

where $\rho(\mathtt{oc}_g^i)$ denotes the estimated cardinality[3] of the guard $\mathtt{oc}_g^i$ and c_r represents the cost of reading a tuple using an index. The values of c_r, c_e, and α are determined experimentally using a set of sample policies and tuples.

The first step in determining $\mathcal{G}(\mathcal{P})$ is to generate all the *candidate guards* ($\mathcal{CG}$), given the object conditions from $\mathcal{P}$, which satisfy the properties of guards as explained in Section 3.2. Different choices of guarded expressions may exist for the same policy given $\mathcal{I}$ and therefore the second step is to select a set of guards from $\mathcal{CG}$ with the goal of minimizing the evaluation cost of $\mathcal{G}(\mathcal{P})$.

4.1 Generating Candidate Guards

Any object condition $\mathtt{oc}_c^l$ in a policy p_l is added to the candidate guard set $\mathcal{CG}$ if it satisfies the properties of a guard i.e., $\mathtt{oc}_c^l.val$ is a constant and $\mathtt{oc}_c^l, attr \in \mathcal{I}$. Each policy $p_l \in \mathcal{P}_{Q_j}$ is guaranteed to have at least one object condition that satisfies these properties (e.g., $\mathtt{oc}_{owner}^l$ or $\mathtt{oc}_{profile}^l$). Guards group together policies and act as a filter reducing the tuples to be evaluated against policies. If only the identical object conditions were to be used as guards, they might group only a small number of policies in their corresponding policy partitions $\mathcal{P}_{G_i}$. This would result in a larger number of guarded expressions in $\mathcal{G}(\mathcal{P})$ of a querier and thus increase the cost of evaluation according to Equation 1.

To improve the grouping capability of a guard, we present an approach which generates additional candidate guards from the already existing candidate guards which have $\mathtt{oc}.val$ as a range of values ($[val_1, val_2]$.). This is done by *merging* together these candidate guards on the same attribute that belong to different policies. For example, consider two policies with the following object conditions on attribute a: $3 \leq a \leq 10$ ($\mathtt{oc}_c^x$) and $4 \leq a \leq 15$ ($\mathtt{oc}_c^y$). Depending on whether it is beneficial to do so, they could be merged to create a new candidate guard $3 \leq a \leq 15$ ($\mathtt{oc}_c^{x \oplus y}$). After merging, this new object condition could be used as a guard for the two policies. The following theorem states the requirement for this merging of object conditions to be beneficial based on their overlap.

THEOREM 1. *Given two object conditions* $\mathtt{oc}_c^x = (attr_c, op_1^x, val_1^x, op_2^x, val_2^x) \in p_x$, $\mathtt{oc}_c^y = (attr_c, op_1^y, val_1^y, op_2^y, val_2^y) \in p_y$ *and* $attr_c \in \mathcal{I}$, *the object condition generated by merging them i.e.,* $\mathtt{oc}_c^{x \oplus y} = (attr_c, op_1, val_1^{x \oplus y}, op_2, val_2^{x \oplus y})$ *with* $val_1^{x \oplus y} = min(val_1^x, val_1^y)$ *and* $val_2^{x \oplus y} = max(val_2^x, val_2^y)$ *is only beneficial if* $[val_1^x, val_2^x] \cap [val_1^y, val_2^y] \neq \phi$.

Proof: Let p_x and p_y be two policies with candidate guards $\mathtt{oc}_c^x$ and $\mathtt{oc}_c^y$, respectively. Based on Equation 3, the cost of evaluating a single policy p_x with $\mathtt{oc}_c^x$ as the guard is

$$cost(p_x) = \rho(\mathtt{oc}_c^x).(c_r + c_e) \tag{4}$$

To simplify the notation in this proof, we use $\mathtt{oc}_c^x$ to denote the values in the range $[val_1^x, val_2^x]$ (similarly for $\mathtt{oc}_c^y$). W.l.o.g. let $min(val_1^x, val_1^y) = val_1^x$ and $max(val_2^x, val_2^y) =$

[3]Estimated using histograms maintained by the DBMS.

val_2^y. If $\mathtt{oc}_c^x \cap \mathtt{oc}_c^y = \emptyset$ the cost of evaluating the merged policy is given by

$$cost(p_x \oplus p_y) = \rho(\mathtt{oc}_c^{x \oplus y}).(c_r + 2.\alpha.c_e) = (\rho(\mathtt{oc}_c^x) + \rho(\mathtt{oc}_c^y)).(c_r + 2.c_e) + \rho(\mathtt{oc}_c^e).(c_r + 2.\alpha.c_e) \quad (5)$$

where $\mathtt{oc}_c^e = (attr_c, op_1, val_2^x, op_2, val_1^y)$ (denotes the extra values that are not covered by either $\mathtt{oc}_c^x$ or $\mathtt{oc}_c^y$). Since $\rho(\mathtt{oc}_c^e) \geq 0$, $cost(p_x \oplus p_y) >= cost(p_x) + cost(p_y)$. Thus, when candidate guards do not overlap, merging them is not beneficial. □

We now check when it is beneficial to merge candidate guards if they overlap i.e., $\mathtt{oc}_c^x \cap \mathtt{oc}_c^y \neq \emptyset$. If these candidate guards were to be merged, the values covered by the merged object condition would be the union of the two ranges, represented by $\mathtt{oc}_c^x \cup \mathtt{oc}_c^y$. The cost of evaluation is given by

$$cost(p_x \oplus p_y) = \rho(\mathtt{oc}_c^x \cup \mathtt{oc}_c^y).(c_r + 2.\alpha.c_e) \quad (6)$$

Applying the inclusion-exclusion principle[4], we have

$$cost(p_x \oplus p_y) = (\rho(\mathtt{oc}_c^x) + \rho(\mathtt{oc}_c^y) - \rho(\mathtt{oc}_c^x \cap \mathtt{oc}_c^y)).(c_r + 2.\alpha.c_e) \quad (7)$$

Given that merging will be beneficial if $cost(p_x \oplus p_y) < cost(p_x) + cost(p_y)$, using Equations 4 and 7 we have the following inequality

$$(\rho(\mathtt{oc}_c^x) + \rho(\mathtt{oc}_c^y) - \rho(\mathtt{oc}_c^x \cap \mathtt{oc}_c^y)).(c_r + 2.\alpha.c_e) < \rho(\mathtt{oc}_c^x).(c_r + c_e) + \rho(\mathtt{oc}_c^y).(c_r + c_e) \quad (8)$$

Simplifying using again the inclusion exclusion principle

$$\frac{\rho(\mathtt{oc}_c^x \cap \mathtt{oc}_c^y)}{\rho(\mathtt{oc}_c^x \cup \mathtt{oc}_c^y)} > \frac{c_e}{c_r + \alpha.c_e} \quad (9)$$

as the right side are all constant values, we can replace it with C. We denote the ratio on the left by $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^y)$. Thus, the merging is beneficial if $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^y) > C$. If this condition is satisfied, we add $\mathtt{oc}_c^{x \oplus y}$ to $\mathcal{P}_x$, $\mathcal{P}_y$, and $\mathcal{CG}$.

As merging is only beneficial, if $|\mathtt{oc}_c^x \cap \mathtt{oc}_c^y| \neq \phi$, we first order the candidate guards by their left range values in the ascending order. Considering *transitive* merges, the number of pair-wise checks to be done between candidate guards could be linear. For instance, consider an additional policy added to the previous example with the following object condition on attribute a, $12 \leq a \leq 18$ ($\mathtt{oc}_c^z$). It is possible that $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^y) < C$ while $\theta(\mathtt{oc}_c^y, \mathtt{oc}_c^z) > C$ and therefore $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^{y \oplus z}) > C$ (i.e., the merged object condition $3 \leq a \leq 18$ is beneficial). The following theorem characterizes when the transitive merges will not be beneficial for candidate guards with certain properties in a $\mathcal{CG}$ sorted in the ascending order of their left range values.

THEOREM 2. *Given three candidate guards $\mathtt{oc}_c^x$, $\mathtt{oc}_c^y$, and $\mathtt{oc}_c^z$ sorted in the ascending order of their left range values with the following properties: $\mathtt{oc}_c^x \cap \mathtt{oc}_c^y \neq \phi$, $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^y) < C$, and $\theta(\mathtt{oc}_c^y, \mathtt{oc}_c^z) > C$. The transitive merge between $\mathtt{oc}_c^x$ and $\mathtt{oc}_c^{y \oplus z}$ will not be beneficial (i.e., $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^{y \oplus z}) < C$) if $\mathtt{oc}_c^x \cap \mathtt{oc}_c^z = \phi$.*

We prove this theorem by contradiction. As $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^y) < C$, using Equation 9 we have

$$\frac{\rho(\mathtt{oc}_c^x \cap \mathtt{oc}_c^y)}{\rho(\mathtt{oc}_c^x \cup \mathtt{oc}_c^y)} < C \quad (10)$$

[4] A ∪ B = A + B - A ∩ B

Note that since $\mathtt{oc}_c^x \cap \mathtt{oc}_c^z = \phi$, $\mathtt{oc}_c^x \cap \mathtt{oc}_c^{y \oplus z} = \mathtt{oc}_c^x \cap \mathtt{oc}_c^y$. Thus, for the transitive merge between $\mathtt{oc}_c^x$ and $\mathtt{oc}_c^{y \oplus z}$ to be beneficial, we should have $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^{y \oplus z}) > C$.

$$\frac{\rho(\mathtt{oc}_c^x \cap \mathtt{oc}_c^y)}{\rho(\mathtt{oc}_c^x \cup \mathtt{oc}_c^y \cup \mathtt{oc}_c^z)} > C \quad (11)$$

This is not possible as the numerator is same in both Equation 10 and Equation 11, while the denominator is larger in Equation 11. □

Given Theorems 1 and 2, the steps for generating $\mathcal{CG}$ from a set of policies $\mathcal{P}$ defined on a relation r_i are: 1) For every $attr_j$ that is part of r_i and has an index defined on it (i.e., $attr_j \in \mathcal{I}$); 2) S_j is the set of all object conditions for all $p^l \in \mathcal{P}$ such that, $\mathtt{oc}_c^l.val$ is a constant and $\mathtt{oc}_c^l.attr = attr_j$; 3) For each S_j containing object conditions with range of values, sort the object conditions by their left values to create a sorted list; 4) For the first candidate guard ($\mathtt{oc}_c^x$) in this sorted list, verify whether the next candidate guard ($\mathtt{oc}_c^y$) is such that $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^y) > C$. If true, then merge both the candidate guards to generate $\mathtt{oc}_c^{x \oplus y}$ which is added to S_j along with p_x and p_y. Else if $\theta(\mathtt{oc}_c^x, \mathtt{oc}_c^y) < C$, then we check if it is beneficial to transitively merge $\mathtt{oc}_c^x$ with the following candidate guard ($\mathtt{oc}_c^z$) using Theorem 2. 5) When transitive merge is no longer beneficial, we move on to the next candidate guard ($\mathtt{oc}_c^y$). The final $\mathcal{CG}$ is constructed by combining all the S_j corresponding to each $attr_j$ that is part of r_i.

4.2 Selecting Guards To Minimize Cost

We next select the set of guards $G \in \mathcal{CG}$ that minimizes the cost of policy evaluation according to Equation 1. The goal of guard selection is to select G from $\mathcal{CG}$ such that every policy in $\mathcal{P}$ is covered exactly once and the cost of evaluation is minimized. We show that this problem is NP-hard, by reducing the well-known weighted Set-Cover problem to it. In the weighted Set-Cover problem, we have a set of elements $E = e_1, \cdots, e_n$ and a set of subsets over E denoted by $S = S_1, \cdots, S_m$ with each set $S_i \in S$ having a weight w_i associated with it. The goal of set cover problem is to select $\min_{\hat{S} \subseteq S} \sum S_i.w_i \mid S_i \in \hat{S}$ and $E = \bigcup_{S_i \in \hat{S}} S_i$. We map E to $\mathcal{P}$, S to $\mathcal{CG}$, and $\hat{S}$ to G. We assign the element e_i to $S_i \in \hat{S}$ when the corresponding policy p_i is assigned to $G_i \in G$. The weight function w_i is set to the read cost of G_i based on using the guard $\mathtt{oc}_c^i$ to read the tuples i.e., $w_i = read\_cost(G_i) = \rho(\mathtt{oc}_c^i).c_r$. If a polynomial time algorithm existed to solve this problem, then it would solve set-cover problem too.

For the purpose of selecting guards that minimize cost of evaluation, we define a utility heuristic which ranks the candidate guards by their benefit per unit read cost (similar to the one used by [19] for optimizing queries with expensive predicates). Each guard $\mathtt{oc}_c^i$, based on its selectivity, reduces the number of tuples that have to be checked against $\mathcal{P}_{G_i}$. The benefit of a guarded expression captures this reduction in evaluation cost for a relation r_i as defined by $benefit(G_i) = c_e.|\mathcal{P}_{G_i}|.(|r_i| - \rho(\mathtt{oc}_c^i))$. Using this benefit method, and the read cost defined earlier, we define the utility of G_i as $utility(G_i) = \frac{benefit(G_i)}{read\_cost(G_i)}$.

Algorithm 1 uses this heuristic to select the best possible guards to minimize the cost of policy evaluation. First, it iterates over $\mathcal{CG}$ and stores each guarded expression $G_i \in \mathcal{CG}$ in a priority queue in the descending order of their utility ($PriorityInsert(Q, G_i, U[i])$. Second, the priority queue

is polled for the G_i with the highest utility ($Extract - Maximum(Q)$). If $\mathcal{P}_{G_i}$ intersects with another $\mathcal{P}_{G_j} \in \mathcal{CG}$, $\mathcal{P}_{G_j}$ is updated to remove the intersection of policies ($\mathcal{P}_{G_i} \cap \mathcal{P}_{G_j}$). After removal, *utility(G_j)* is recomputed and the updated G_j is reinserted into priority queue in the order of its utility. The result is thus the subset of candidate guards (G) that covers all the policies in $\mathcal{P}$ and minimizes *cost($\mathcal{G(P)}$)* as in Equation 1.

Algorithm 1 Selection of guards.

```
1: function GuardSelection(CG)
2:     for i in 1 ··· |CG| do
3:         C[i] = cost(G_i); U[i] = utility(G_i)
4:     Q ← φ
5:     for i in 1 ··· |CG| do
6:         PriorityInsert(Q, G_i, U[i])
7:     while Q is not empty do
8:         G_max = Extract-Maximum(Q); G ← G_max
9:         for each G_i in Q do
10:            if P_{G_i} ∩ P_{G_max} ≠ φ then
11:                P_{G_i} = P_{G_i} \ P_{G_max}; Remove(Q, G_i)
12:                if P_{G_i} ≠ φ then
13:                    B = benefit(G_i); U[i] = B / C[i]
14:                    PriorityInsert(Q, G_i, U[i])
       return G
```

5. IMPLEMENTING SIEVE

Sieve[5] is a general-purpose middleware that intercepts queries posed to a DBMS, optimally rewrites them, and submits the queries back to the underlying DBMS for efficient execution that is compliant with the access control policies. In this section, we first present the rewrite approach with *guarded expressions* in DBMSs. Then, we present two optimization techniques to improve this rewrite by utilizing policy evaluation operator and query predicates. Finally, we illustrate a sample rewritten query in Sieve.

5.1 Query Rewrite with Guarded Expressions

Our goal is to evaluate policies for query Q_i by replacing any relation $r_j \in Q_i$ by a projection of r_j that satisfies the guarded policy expression $\mathcal{G}(\mathcal{P}_{r_j})$ where $\mathcal{P}_{r_j}$ is the set of policies defined for the specific querier, purpose, and relation. To this end, we first use the `WITH` clause for each relation $r_j \in Q_i$ that selects tuples in r_j satisfying the guarded policy expression. Using the `WITH` clause, the policy check is performed only once even if the same relation appears multiple times in the query. This rewritten query replaces every occurrence of r_j with the corresponding $\hat{r_j}$.

```
WITH r̂_j AS (
  SELECT * FROM r_j WHERE G_1 OR G_2 OR ··· OR G_n)
```

Optimizers might choose sub-optimal plans when executing the complex Sieve rewritten queries. Sieve utilizes DBMS extensibility features (e.g., index usage hints[6], optimizer explain[7], UDFs) offered by DBMSs that allows it to suggest index plans to the underlying optimizer. Since such features vary across DBMSs, guiding optimizers requires a platform dependent connector that can rewrite the query appropriately. In systems such as MySQL, Oracle, DB2, and SQL Server that support index usage hints, Sieve can rewrite the query to explicitly force indexes on guards. For example, in MySQL using `FORCE INDEX` hints, which tell the optimizer that a table scan is expensive and should only be used if the DBMS cannot use the suggested index to find rows in the table, the rewritten query will be as follows:

```
WITH r̂_j AS (
  SELECT * FROM r_j [FORCE INDEX (oc_g^1)] WHERE G_1
      UNION
  SELECT * FROM r_j [FORCE INDEX (oc_g^2)] WHERE G_2
      UNION···
  SELECT * FROM r_j [FORCE INDEX (oc_g^n)] WHERE G_n)
```

Some systems, like PostgreSQL, do not support index hints explicitly. In such cases, Sieve still does the above rewrite (without index usage hints) but depends upon the underlying optimizer to select appropriate indexes.

5.2 Policy Evaluation Operator

For a given guarded expression G_i, a tuple that satisfies its guard $\texttt{oc}_g^i$ is checked against its policy partition $\mathcal{P}_{G_i}$. We define this strategy of evaluating policies inline with a guard as $Guard\&Inlining$. This evaluation strategy could be expensive depending upon the number of policies in $\mathcal{P}_{G_i}$. We introduce an alternative strategy which uses the policy evaluation operator $\Delta(G_i, \texttt{QM}^i, t_t)$ as an alternative to evaluating policies inline with a guard. This operator retrieves only the policies that are applicable to a tuple t_t (that satisfied $\texttt{oc}_g^i$ of G_i) based on G_i, the query metadata $\texttt{QM}^i$, and tuple context of t_t. We call this strategy of using the policy evaluation operator in conjunction with guards as $Guard\&\Delta$. Sieve adaptively chooses between these two strategies depending on the number of policies in the guard partition (i.e., $|\mathcal{P}_{G_i}|$).

The policy evaluation operator $\Delta(G_i, \texttt{QM}^i, t_t)$ (which is part of $Guard\&\Delta$) is implemented using User Defined Functions (UDFs) supported by DBMSs. This implementation is done per $r_j \in \mathcal{R}$ as the tuple context of t_t used to retrieve policies varies per relation. The policy evaluation operator performs two operations. First, it retrieves a subset of $\mathcal{P}_{G_i}$ which only includes the relevant policies based on the query metadata $\texttt{QM}^i$ and the tuple context of t_t. The tuple context is determined by its values for different attributes (e.g., $r_i.owner$) and $\texttt{QM}^i$ is information associated with the query Q_i such as $\texttt{QM}^i_{querier}$ and $\texttt{QM}^i_{purpose}$. Second, given such a subset, $\bar{\mathcal{P}}_{G_i}$, the operator evaluates each policy in it, $p_i \in \bar{\mathcal{P}}_{G_i}$, on t_t using the access control semantics defined in Section 3.1. An example invocation of the $\Delta(G_i, \texttt{QM}^i, t_t)$ is as follows: *delta(32, "Prof.Smith", "Analysis", "owner", "ts-date", "ts -time", "wifiAP")*. The first parameter, *32*, denotes the id of the persisted guarded expression G_i in the database. The second set of parameters {*"Prof.Smith", "Analysis"*} belong to the metadata of the query $\texttt{QM}^i$. The final set of parameters (*"owner", "ts-date", "ts -time", "wifiAP"*) denote the attributes of the tuple and thus defines its context.

Sieve uses a cost model to compare between the two different strategies ($Guard\&\Delta$ and $Guard\&Inlining$) and chooses the best one for performing the rewrite. This comparison is performed for each guarded expression G_i in a guarded policy expression $\mathcal{G(P)}$ (along with query metadata). We model the cost of each strategy by computing the cost of evaluating policies per tuple since the number of tuples to check are the same in both cases. The cost of the $Guard\&\Delta$ strategy

[5]The implementation of Sieve (with connectors for both MySQL and PostgreSQL) is available at `https://github.com/primalpop/sieve`.

[6]`https://dev.mysql.com/doc/refman/8.0/en/index-hints.html`

[7]`https://www.postgresql.org/docs/13/sql-explain.html`

is estimated by the invocation and execution cost of the UDF implementation of policy evaluation operator. Thus, *cost(Guard&Δ)* $= UDF_{inv} + UDF_{exec}$ where the UDF_{inv} and UDF_{exec} represent the cost of invocation and execution of the UDF, respectively. We compute this cost by executing the UDF with different guards (with different number of policies in their partitions) and averaging across them. As both the terms involved are constants, *cost(Guard&Δ)* does not vary across guarded expressions.

The cost of *Guard&Inlining* is determined by the number of policies in the policy partition of the guard i.e., $|\mathcal{P}_{G_i}|$. Thus, *cost(Guard&Inlining)*$=\alpha.|\mathcal{P}_{G_i}|.c_e$ (based on Equation 2). Unlike *cost(Guard&Δ)*, this cost is not a constant and varies depending upon the guarded expression. Therefore, after generating guarded expressions Sieve computes *cost(Guard&Inlining)* and compares it against the pre-computed *cost(Guard&Δ)* to determine the appropriate rewriting for each guarded expression. Our experiments (see Section 6) indicate that the usage of the *Guard&Δ* strategy is beneficial if $|\mathcal{P}_{G_i}| > 120$.

5.3 Exploiting Selective Query Predicates

In the query rewrite strategy with guarded expressions presented in Section 5.1, we used the guards to read the relevant tuples using an index. This is followed by evaluating against policies using inlining or policy evaluation operator. We now consider the situation where the selection predicates that appear in Q_i are highly selective and could be exploited to read the tuples using an index on them instead of on the guards. Sieve considers the following two strategies for reading tuples using the index. 1) Using guards followed by evaluation of the policy partitions associated with them (referred to as *IndexGuards*); 2) Using the query predicate $Q_i.pred$ in Q_i followed by the evaluation of the guarded policy expression (referred to as *IndexQuery*). Each of these strategies use guarded expressions to evaluate the policies and generate $\hat{r_j}$ from r_j on which Q_i is evaluated.

Sieve uses a cost model to compare between these two strategies (*IndexGuards* and *IndexQuery*) and chooses the best one for performing the final rewrite. This comparison is done per query Q_i. The cost of *IndexQuery* is determined by using the query explain feature of DBMSs for Q_i, which returns the query predicate $Q_i.pred$ in Q_i used for reading tuples using the index (if any) and its estimated selectivity. We use this to compute *cost(IndexQuery)* $= \rho(Q_i.pred).c_r$. If index is not used for access, we set *cost(IndexQuery)* $= \infty$. For *IndexGuard*, Sieve estimates *cost(IndexGuards)* $= \sum_{G_i \in G} \rho(\mathtt{oc}_g^i).c_r$ where $\mathtt{oc}_g^i$ is the guard used in the guarded expression G_i. Note that this is an upper bound of the cost for reading tuples using index as it does not consider any optimizations such as index merge. Sieve chooses the best strategy, at query execution time, by comparing their costs (*cost(IndexGuards)* vs. *cost(IndexQuery)*). If the *IndexGuards* strategy is chosen, we use the rewrite illustrated in Section 5.1. Otherwise, if *IndexQuery* is selected, we use index usage hints with $Q_i.pred$ (instead of $\mathtt{oc}_g^i$).

5.4 Sieve generated Query Rewrite

Using the different strategies presented, we now revisit the query in Section 2.1 to study the tradeoff between student performance and attendance to classes. A possible rewrite by Sieve to evaluate the policies defined on $WifiDataset$ and generate $WiFiDatasetPol$ is as follows:

```
WITH WiFiDatasetPol AS (
 SELECT * FROM WiFiDataset as W FORCE INDEX(oc_g^1)
  WHERE (oc_g^1 AND (G_1))
 UNION
 SELECT * FROM WiFiDataset as W FORCE INDEX(oc_g^2)
  WHERE (oc_g^1 AND (G_2))
 ....
 UNION
 SELECT * FROM WiFiDataset as W FORCE INDEX(oc_g^n)
  WHERE (oc_g^n AND delta(32,"Prof.Smith", "Analysis","
      owner","ts-date", "ts-time", "wifiAP")=true)
) StudentPerf(WifiDatasetPol, Enrollment, Grades)
```

The `WiFiDatasetPol` replaces `WiFiDataset` in the original query. This rewrite includes the set of guards generated for the querier (Prof. Smith) and his purpose (Analysis) given the policies defined for him. As Sieve selected the *IndexGuards* strategy, we use the index usage hints on guards (through the `FORCE INDEX` command since this rewrite is for MySQL) as explained in Section 5.3. Finally, for one specific guarded expression (G_n) Sieve selected the *Guard&Δ* strategy (see Section 5.2). Hence, its policy partition was replaced by the call to the UDF that implements the Δ operator.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Datasets. We used the *TIPPERS* dataset [26] consisting of connectivity logs generated by the 64 WiFi Access Points (APs) at the Computer Science building at UC Irvine for a period of three months. These logs are generated when a WiFi enabled device (e.g., a smartphone or tablet) connects to one of the WiFi APs and contain the hashed identification of the device's MAC, the AP's MAC, and a timestamp. The dataset comprises 3.9M events corresponding to 36K different devices (the signal of some of the WiFi APs bleeds to outside the building and passerby devices/people are also observed). This information can be used to derive the occupancy levels in different parts of the building and to provide diverse location-based services (see Section 2.1) since device MACs can be used to identify individuals. Since location information is privacy-sensitive, it is essential to limit access to this data based on individuals' preferences.

We also used a synthetic dataset containing WiFi connectivity events in a shopping mall for scalability experiments with even larger number of policies. We refer to this dataset as *Mall*. We generated the *Mall* dataset using the IoT data generation tool in [18] to generate synthetic trajectories of people in a space (we used the floorplan of a mall extracted from the Web) and sensor data based on those. The dataset contains 1.7M events from 2,651 different devices representing customers.

Queries. We used a set of query templates based on the recent IoT SmartBench benchmark [18] which include a mix of analytical and real-time tasks and target queries about (group of) individuals. Specifically, query templates Q_1 - Retrieve the devices connected for a list of locations during a time period (e.g., for location surveillance); Q_2 - Retrieve devices connected for a list of given MAC addresses during a time period (e.g., for device surveillance); Q_3 - Number of devices from a group or profile of users in a given location (e.g., for analytic purposes). Based on these templates, we generated queries at three different selectivities

(low, medium, high) by modifying configuration parameters (locations, users, time periods). Below, when referring to a particular query type (i.e., Q_1, Q_2, or Q_3), we mean the set of queries generated for such type.

Policy Generation. The TIPPERS dataset, collected for a limited duration with special permission from UC Irvine for the purpose of research, does not include user-defined policies. We therefore generated a set of synthetic policies. As part of the TIPPERS project, we conducted several town hall meetings and online surveys to understand the privacy preferences of users about sharing their WiFi-based location data. The surveys, as well as prior research [22, 24], indicate that users express their privacy preferences based on different user profiles (e.g., students for faculty) or groups (e.g., my coworkers, classmates, friends, etc.). Thus, we used a profile-based approach to generate policies specifying which events belonging to individual can be accessed by a given querier (based on their profile) for a specific purpose in a given context (e.g., location, time).

We classified devices in the TIPPERS dataset as belonging to users with different profiles (denoted by *profile(u_k)* for user u_k) based on the total time spent in the building and connectivity patterns. Devices which rarely connect to APs in the building (i.e., less than 5% of the days) are classified as *visitors*. The non-visitor devices are then classified based on the type of rooms they spent most time in: *staff* (staff offices), *undergraduate students* (classrooms), *graduate students* (labs), and *faculty* (faculty offices). As a result, we classified 31,796 visitors, 1,029 staff, 388 faculty, 1,795 undergraduate, and 1,428 graduate from a total of 36,436 unique devices in the dataset. Our classification is consistent with the expected numbers for the population of the monitored building. We also grouped users into groups based on the affinity of their devices to rooms in the building which is defined in terms of time spent in each region per day. Thus, each device is assigned to a group with maximum affinity. In total, we generated 56 groups with an average of 108 devices per group.

We define two kinds of policies based on whether they are an unconcerned user or an advanced user as described in Section 2.1. Unconcerned users subscribe to the default policies set by administrator which allows access to their data based on user-groups and profiles. Advanced users define on average 40 policies, given the large number of control options (such as device, time, groups, profiles, and locations) in our setting. In total, the policy dataset generated contains 869,470 policies with each individual defining 472 policies on average and appearing as querier in 188 policies defined by others on average. The above policies are defined to allow access to data in different situations. Any other access that is not captured by the previous policies will be denied (based on the default opt-out semantics defined in Section 3.1).

For the *Mall* dataset, the shops were categorized into six types based on the services they provide, such as arcade or movies. We also classified customers into regular and irregular, based on their shop visits. For each customer, we then defined two types of policies depending on whether they were regular or irregular. Regular customers allowed shops they visit the most to have access to their location during open hours. Irregular customers shared their data only with specific shop types depending on whether there were sales or discounts. Finally, if a customer expressed an interest in a particular shop category, we also generated policies which allowed access of their data to the shops in the category for a short period of time (e.g., lightning sales). In total, this policy dataset generated on top of *Mall* dataset contains 19,364 policies defined for 35 shops (queriers) in the mall with 551 policies on average per shop.

Database System. We ran the experiments on an individual machine (CentOS 7.6, Intel(R) Xeon(R) CPU E5-4640, 2799.902 Mhz, 20480 KB cache size) in a cluster with a shared total memory of 132 GB. We performed experiments on MySQL 8.0.3 with InnoDB as it is an open source DBMS which supports index usage hints. We configured the *buffer_pool_size* to 4 GB. We also performed experiments on PostgreSQL 13.0 with *shared_buffers* configured to 4 GB.

6.2 Experimental Results

We first study the performance of the guarded expression generation algorithm (Experiment 1). Then, we validate the design choices in Sieve (Experiment 2) and compare the performance of Sieve against the baselines (Experiment 3). The previous experiments are performed on the MySQL system. Next, we study the performance of Sieve on PostgreSQL which, in contrast to MySQL, does not support index usage hints (Experiment 4). In the final experiment, we stress test our approach with a very large number of policies (Experiment 5). The first four experiments use the *TIPPERS* dataset and the final experiment the *Mall* dataset.

Experiment 1: Cost for generating Guarded Expressions and Effectiveness. The goal of this experiment is to study the cost of generating guarded expressions for a querier, as factor of the number of policies, and the quality of generated guards. To analyze the cost of guarded expression generation, we generate guarded expressions for all the users using the algorithm described in Section 4 and collect the generation times in a set. We sort these costs (in milliseconds) and average their value in groups of 50 users showing the result in Figure 2. The cost of guard generation increases linearly with number of policies. As guarded expression generation is also dependent on the selectivity of policies, number of candidate guards generated, which is also a factor of overlap between predicates, we sometimes observe a slight decrease in the time taken with increasing policies. The overhead of the cost of generating guarded expression is minimal, for instance, the cost of generating a guard for a querier with 160 policies associated (e.g., the student trying to locate classmates explained in Section 2.1) is around 150ms.

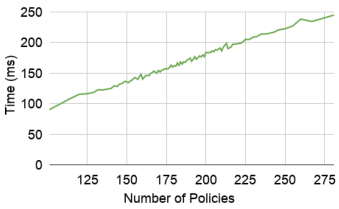

Figure 2: Guard generation cost.

Table 2: Analysis of policies and generated guards.

	min	avg	max	SD
$\lvert\mathcal{P}_{u_k}\rvert$	31	187	359	38
$\lvert G\rvert$	2	31	60	10
$\lvert\mathcal{P}_{G_i}\rvert$	4	7	60	5
$\rho(G_i)$	0.01%	3%	24%	2%
Savings	0.99	0.99	1	7e−4

Table 3: Analysis of number of guards and total cardinality.

$\rho(G)$	$\lvert G\rvert$	
	low	high
low	227.2	537.0
high	469.0	1,406.7

We present the results of the analysis of the policies and generated guarded expressions in Table 2. Each user is affected, on average, by 187 policies ($|\mathcal{P}_{u_k}|$). This number

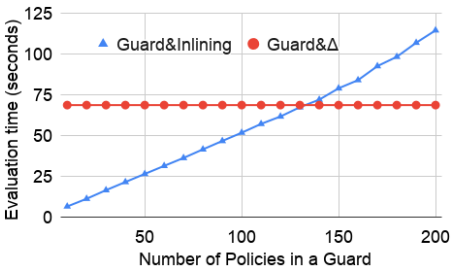

Figure 3: *Inlining* vs. Δ.

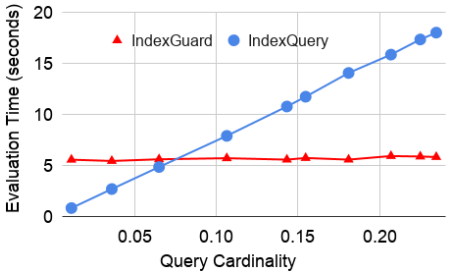

Figure 4: Index choice.

depends on their profiles (e.g., student) and group memberships. Sieve creates an average of 31 guards per user with the mean partition cardinality (i.e., $|\mathcal{P}_{G_i}|$) as 7. The total cardinality of guards in the guarded expression is low (i.e., $\rho(G_i)$) which helps in filtering out tuples before performing policy evaluation. In cases with high cardinality guards (e.g., maximum of 24%), Sieve will not use force an index scan in that particular guard as explained in Section 5. Savings is computed as ratio of the difference between total number of policy evaluations without and with using the guard and the number of policy evaluations. This was computed on a smaller sample of the entire dataset and the results show that guards help in eliminating around 99% of the policy checks compared to policy evaluation.

Experiment 2.1: Inline vs. Operator Δ. SIEVE uses a cost model to determine for each guard whether to inline the policies or to evaluate the policies using the Δ operator. The Δ operator has an associated overhead of UDF invocation but it can utilize the tuple context to reduce the number of policies that need to be checked per tuple. For the purpose of studying this tradeoff in both inlining and using the Δ operator, we gradually increased the number of policies that are part of the partition of a guard and observed the cost of policy evaluation. As expected, we observed that when the number of policies are about 120, the cost of UDF invocations is amortized by the savings from filtering policies by the Δ operator (see Figure 3).

Experiment 2.2: Query Index vs. Guard Index. In Sieve, we use a cost model to choose between using the *IndexQuery* and *IndexGuards* as explained in Section 5. We evaluated this cost model by analyzing the cost of evaluation against increasing query cardinality for three different guard cardinalities (low, medium, high). Figure 4 shows the results averaged across these three guard cardinalities. As expected, at low query cardinality it is better to utilize *IndexQuery*, while at medium and high query cardinalities (> 0.07), *IndexGuards* are the better choice. Note that in both these options, guarded expressions are used as filter on top of the results from Index Scan.

Table 4: Overall performance for $Q1$, $Q2$, and $Q3$ (in ms).

	$\rho(Q)$	$Baseline_P$	$Baseline_I$	$Baseline_U$	Sieve
$Q1$	low	1,668	906	9,122	418
	mid	15,356	910	$23{,}575^+$	453
	high	TO	937	TO	523
$Q2$	low	860	916	7,787	407
	mid	7,191	922	$22{,}617^+$	454
	high	$29{,}765^+$	962	TO	475
$Q3$	low	883	881	14,379	477
	mid	2,217	2,209	TO	476
	high	3,502	3,543	TO	521

Experiment 3: Query Evaluation Performance. We compare the performance of Sieve (implemented as detailed in Section 5) against three different baselines. In the first baseline, $Baseline_P$, we append the policies that apply to the querier to the `WHERE` condition of the query. Second, $Baseline_I$, performs an index scan per policy (forced using index usage hints) and combines the results using the `UNION` operator. Third, $Baseline_U$ is similar to $Baseline_P$ but instead of using the policy expression, it uses a UDF defined on the relation to evaluate the policies. The UDF takes as input all the attributes of the tuple. $Baseline_U$ significantly reduces the number of policies to be evaluated per tuple and is therefore an interesting optimization strategy for low cardinality queries. UDF invocations are expensive, so it might be preferable to execute the UDF as late as possible from the optimization perspective [19]. To preserve correctness of policy enforcement as defined in Section 3.1, UDF operations have to be performed before any non-monotonic query operations.

For each of the query types ($Q1$, $Q2$, Q_3), we generate a workload of queries with three different selectivity classes posed by five different queriers of belonging to four different profiles. The values chosen for these three selectivity classes (low, medium, high) differed depending upon the query type. We execute each query along with the access control mechanism 5 times and average the execution times. The experimental results below give the average warm performance per query. The time out was set at 30 seconds. If a strategy timed out for all queries of that group we show the value TO. If a strategy timed out for some of the queries in a group but not all, the table shows the average performance only for those queries that were executed to completion; those time values are denoted as t^+.

Table 4 shows the average performance for the three query types. The performance of $Baseline_P$ and $Baseline_U$ degrades with increasing cardinality of the associated query as they rely on the query predicate for reading the tuples. The relative reduction in overhead for Q3 for $Baseline_P$ at high cardinalities is because the optimizer is able to use the low cardinality join condition to perform a nested index loop join. The performance of Sieve and $Baseline_I$ remains the same across query cardinalities as they utilize the policy and guard predicates for reading the tuples and hence are not affected by the query cardinality. The increase in the speedup between these two sets of approaches clearly demonstrate that exploiting indices paid off. For $Baseline_P$, the optimizer is not able to exploit indices at high cardinalities and resorts to performing linear scan. In $Baseline_U$, the cost of UDF invocation per tuple far outweighed any benefits from filtering of policies. $Baseline_I$, generated by careful rewriting with an index scan per policy, performs significantly better than the previous two baselines. The performance degrade of $Baseline_I$ for Q3 is due to the optimizer

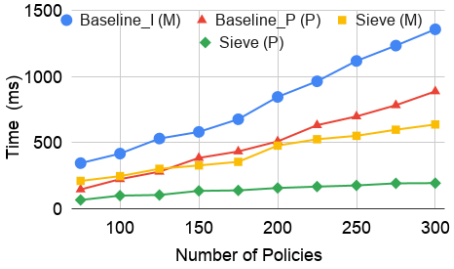

Figure 5: Sieve on MySQL and PostgreSQL.

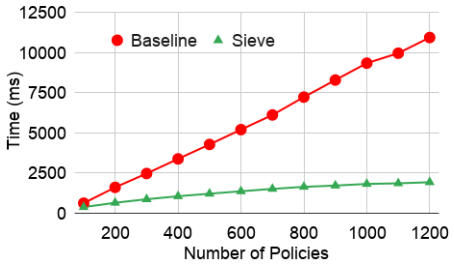

Figure 6: Scalability comparison.

preferring to perform the nested loop join first instead of the index scans. In comparison to all these baselines, Sieve is significantly faster at all different query cardinalities.

Experiment 4: Sieve on PostgreSQL. In the previous experiments we used MySQL, which supports hints for index usage, thus enabling SIEVE to explicitly force the optimizer to choose guard indexes. However, other DBMSs, such as PostgreSQL, do not support index usage hints explicitly (as discussed in Section 5.1). To study Sieve's performance in such systems, we implemented a Sieve connector to PostgreSQL using the same rewrite strategy but without index usage hints. To have a cumulative set of policies (i.e., the larger set of policies contain the smaller set of policies) for evaluation, we chose 5 queriers to whom at least 300 policies apply. For each querier, we divided their policies into 10 different sets of increasing number of policies starting with smallest set of 75 policies. The order and the specific policies in these sets were varied 3 times by random sampling. The results in Figure 5 shows the average performance of different strategies for each set size averaged across queriers and the samples for `SELECT ALL` queries.

The four strategies tested in this experiment are: the best performing baseline for MySQL ($Baseline_I$(M)), the baseline in PostgreSQL ($Baseline_P$(P)), and Sieve in both MySQL and PostgreSQL (Sieve (M) and Sieve (P)). The results show that not only Sieve outperforms the baseline in PostgreSQL but also the speedup factor w.r.t. the baseline is even higher than in MySQL. Additionally, the speedup factor in PostgreSQL is the highest at largest number of policies. Based on our analysis of the query plan chosen by PostgreSQL, it correctly chooses the guards for performing index scans (as intended by Sieve) even without the index usage hints. In addition, PostgreSQL supports combining multiple index scans by preparing a bitmap in memory. It used these bitmaps to *OR* the results from the guards whenever it was possible, and the only resultant table rows are visited and obtained from the disk. With a larger number of guards (for larger number of policies), PostgreSQL was also able to more efficiently filter out tuples compared to using the policies. Thus, Sieve benefits from reduced number of disk reads (due to bitmap) as well as a smaller number of evaluations against the partition of the guarded expression.

Experiment 5: Scalability. The previous experiment shows that the speedup of Sieve w.r.t. the baselines increases with an increasing number of policies, especially for PostgreSQL. We explore this aspect further on PostgreSQL using the *Mall* dataset where the generation of very large number of policies per querier (in this case the querier is a shop) is more feasible as we can simulate more customers. We used the same process than in Experiment 4 to generate cumulative set of policies by choosing 5 queriers/shops with at least 1,200 policies defined for them. Figure 6 reaffirms how the speedup of Sieve compared against the baseline increases linearly starting from a factor of 1.6 for 100 policies to a factor of 5.6 for 1,200 policies. We analyzed the query plan selected by the optimizer for the Sieve rewritten queries. We observed that with larger number of guards, PostgreSQL is able to utilize the bitmaps in memory to gain additional speedups from guarded expressions (as explained in Experiment 4). Also, this experiment shows that Sieve outperforms the baseline for a different dataset which shows the generality of our approach.

7. CONCLUSION

In this paper, we presented Sieve, a layered approach to enforcing large number of fine-grained policies during query execution. Sieve combines two optimizations: reducing the number of policies that need to be checked against each tuple, and reducing the number of tuples that need to be checked against complex policy expressions. Sieve is designed as a general purpose middleware approach and we have layered it on two different DBMSs. The experimental evaluation, using a real dataset and a synthetic one, highlights that Sieve enables existing DBMSs to perform efficient access control. Sieve significantly outperforms existing strategies for implementing policies based on query rewrite. The speedup factor increases with increasing number of policies and Sieve's query processing time remains low even for thousands of policies per query.

We plan to leverage the experience gained while developing Sieve to pursue a tighter integration with the database query optimizer. Also, we plan to study possible mechanisms to combine certain amount of preprocessing at insertion time to simplify policy checking and guard evaluation at query time to extend Sieve to co-optimize a query and policy workload.

Acknowledgement

This material is based on research sponsored by DARPA under agreement number FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation there on. The views and conclusions contained here in are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is partially supported by the NSF grants 2032525, 1545071, and 1527536. Our thanks to the members of TIPPERS research group for discussions and feedback. We would also like to thank the reviewers to their detailed comments.

8. REFERENCES

[1] California consumer privacy act CCPA. https://oag.ca.gov/privacy/ccpa. [Online; accessed 1-June-2020].

[2] California online privacy protection act CalOPPA. https://leginfo.legislature.ca.gov/faces/codes_displaySection.xhtml?lawCode=BPC&sectionNum=22575. [Online; accessed 1-June-2020].

[3] General data protection regulation GPDR. https://gdpr.eu/. [Online; accessed 1-June-2020].

[4] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *21st Int. Conf. on Data Engineering*, pages 1013–1022, 2005.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *PVLDB*, pages 143–154, 2002.

[6] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster. Keeping the smart home private with smart(er) iot traffic shaping. *PoPETs*, 2019(3):128–148, 2019.

[7] E. Bertino. Data security and privacy in the iot. In *19th Int. Conf. on Extending Database Technology*, pages 1–3, 2016.

[8] E. Bertino, G. Ghinita, A. Kamra, et al. Access control for databases: concepts and systems. *Foundations and Trends in Databases*, 3(1–2):1–148, 2011.

[9] J.-W. Byun, E. Bertino, and N. Li. Purpose based access control of complex data for privacy protection. In *10th ACM Symposium on Access Control Models and Technologies*, pages 102–110, 2005.

[10] J.-W. Byun and N. Li. Purpose based access control for privacy protection in relational database systems. *PVLDB*, 17(4):603–619, 2008.

[11] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *ACM SIGMOD Int. Conf. on Management of data*, pages 361–372, 2003.

[12] P. Colombo and E. Ferrari. Enforcement of purpose based access control within relational database management systems. *IEEE Transactions on Knowledge and Data Engineering*, 26(11):2703–2716, 2014.

[13] P. Colombo and E. Ferrari. Efficient enforcement of action-aware purpose-based access control within relational database management systems. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2134–2147, 2015.

[14] P. Colombo and E. Ferrari. Fine-grained access control within nosql document-oriented datastores. *Data Science and Engineering*, 1(3):127–138, 2016.

[15] P. Colombo and E. Ferrari. Towards a unifying attribute based access control approach for nosql datastores. In *33rd Int. Conf. on Data Engineering*, pages 709–720, 2017.

[16] P. Colombo and E. Ferrari. Access control enforcement within mqtt-based internet of things ecosystems. In *23nd ACM Symposium on Access Control Models and Technologies*, pages 223–234, 2018.

[17] P. Colombo and E. Ferrari. Access control technologies for big data management systems: literature review and future trends. *Cybersecurity*, 2(1), 2019.

[18] P. Gupta, M. J. Carey, S. Mehrotra, and R. Yus. Smartbench: A benchmark for data management in smart spaces. In *PVLDB*, volume 13, 2020.

[19] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems (TODS)*, 23(2):113–157, 1998.

[20] J. Heo, H. Lim, S. B. Yun, S. Ju, S. Park, and R. Lee. Descriptive and predictive modeling of student achievement, satisfaction, and mental health for data-driven smart connected campus life service. In *9th Int. Conf. on Learning Analytics & Knowledge*, 2019.

[21] M. Langheinrich. Privacy by design—principles of privacy-aware ubiquitous systems. In *3rd Int. Conf. on Ubiquitous Computing*, pages 273–291, 2001.

[22] H. Lee and A. Kobsa. Privacy preference modeling and prediction in a simulated campuswide iot environment. In *IEEE Int. Conf. on Pervasive Computing and Communications, PerCom 2017*, pages 276–285, 2017.

[23] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *PVLDB*, pages 108–119, 2004.

[24] J. Lin, B. Liu, N. Sadeh, and J. I. Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *10th USENIX Conf. on Usable Privacy and Security*, pages 199–212, 2014.

[25] K. Loney. *Oracle Database 11g The Complete Reference*. McGraw-Hill, Inc., 2008.

[26] S. Mehrotra, A. Kobsa, N. Venkatasubramanian, and S. R. Rajagopalan. TIPPERS: A privacy cognizant iot environment. In *2016 IEEE Int. Conf. on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6, 2016.

[27] R. V. Nehme, H.-S. Lim, and E. Bertino. Fence: Continuous access control enforcement in dynamic data stream environments. In *3rd ACM Conf. on Data and Application Security and Privacy*, pages 243–254, 2013.

[28] N. Panwar, S. Sharma, G. Wang, S. Mehrotra, and N. Venkatasubramanian. Verifiable round-robin scheme for smart homes. In *9th ACM Conf. on Data and Application Security and Privacy, CODASPY*, pages 49–60, 2019.

[29] P. Pappachan, M. Degeling, R. Yus, A. Das, S. Bhagavatula, W. Melicher, P. E. Naeini, S. Zhang, L. Bauer, A. Kobsa, et al. Towards privacy-aware smart buildings: Capturing, communicating, and enforcing privacy policies and preferences. In *IEEE 37th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, pages 193–198, 2017.

[30] P. Pappachan, R. Yus, S. Mehrotra, and J.-C. Freytag. Sieve: A Middleware Approach to Scalable Access Control for Database Management Systems. *arXiv e-prints*, page arXiv:2004.07498, July 2020.

[31] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *ACM SIGMOD Int. Conf. on*

Management of data, pages 551–562, 2004.

[32] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modification. In *1974 Annual Conf.*, pages 180–186, 1974.

[33] G. Sun, V. Chang, M. Ramachandran, Z. Sun, G. Li, H. Yu, and D. Liao. Efficient location privacy algorithm for internet of things (iot) services and applications. *J. Network and Computer Applications*, 89:3–13, 2017.

[34] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun. On the correctness criteria of fine-grained access control in relational databases. In *PVLDB*, pages 555–566, 2007.