GuardSpark++: Fine-Grained Purpose-Aware Access Control for Secure Data Sharing and Analysis in Spark

Tao Xue
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
xuetao@iie.ac.cn

Yu Wen*
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
wenyu@iie.ac.cn

Bo Luo
The University of Kansas
Kansas, USA
bluo@ku.edu

Boyang Zhang
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
zhangboyang@iie.ac.cn

Yang Zheng
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
zhengyang@iie.ac.cn

Yanfei Hu
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
huyanfei@iie.ac.cn

Yingjiu Li
Department of Computer and Information Science, University of Oregon
Oregon, USA
yjli@smu.edu.sg

Gang Li
Centre for Cyber Security Research and Innovation, Deakin University
Geelong, Australia
gang.li@deakin.edu.au

Dan Meng
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
mengdan@iie.ac.cn

ABSTRACT

With the development of computing and communication technologies, extremely large amount of data has been collected, stored, utilized, and shared, while new security and privacy challenges arise. Existing platforms do not provide flexible and practical access control mechanisms for big data analytics applications. In this paper, we present GuardSpark++, a fine-grained access control mechanism for secure data sharing and analysis in Spark. In particular, we first propose a purpose-aware access control (PAAC) model, which introduces new concepts of data processing/operation purposes to conventional purpose-based access control. An automatic purpose analysis algorithm is developed to identify purposes from data analytics operations and queries, so that access control could be enforced accordingly. Moreover, we develop an access control mechanism in Spark Catalyst, which provides unified PAAC enforcement for heterogeneous data sources and upper-layer applications. We evaluate GuardSpark++ with five data sources and four structured data analytics engines in Spark. The experimental results show that GuardSpark++ provides effective access control functionalities with a very small performance overhead (average 3.97%).

CCS CONCEPTS

• **Security and privacy → Access control**.

KEYWORDS

Spark, big data, access control, data sharing, data protection, purpose

ACM Reference Format:
Tao Xue, Yu Wen, Bo Luo, Boyang Zhang, Yang Zheng, Yanfei Hu, Yingjiu Li, Gang Li, and Dan Meng. 2020. GuardSpark++: Fine-Grained Purpose-Aware Access Control for Secure Data Sharing and Analysis in Spark. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427640

*Corresponding Author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC 2020, December 7–11, 2020, Austin, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427640

1 INTRODUCTION

In the big data era, tremendous amount of data is being collected, stored, and utilized in various platforms and applications [29]. The big data processing platforms have been developed to harness data, to facilitate data analytics functions, and to discover intrinsic value

from data [2, 28, 33, 43, 74]. They were designed to access various data sources, support hybrid data analytics engines (e.g., advanced SQL, machine learning, and graph) and efficiently process a very large volume of data. However, data security and privacy were not sufficiently considered when these platforms were designed. As a result, very limited data protection functions are provided, so that data owners and administrators are unable to specify fine-grained or complex access control intentions. For example, users often get full access to sensitive raw data when they are supposed to only execute aggregate or statistical queries. Moreover, data gets more vulnerable in the context of large-scale cross-organizational data sharing [10, 20], where curious users may correlate data/attributes from multiple sources or platforms to further extract sensitive information.

In data management and sharing applications, access control is the essential protection mechanism to defend against unauthorized access to sensitive data [40, 41, 44, 70]. A baseline solution for the big data platforms is to directly employ access control functionalities provided by the underneath data source or OS. However, this simple solution is usually insufficient in supporting users' access control needs: 1) those mechanisms often fall short in providing fine-grained (attribute-, record-, or cell-level) access control capabilities, which is the *de facto* granularity standard for state-of-art data management applications [68]. For instance, the Hadoop Distributed File System (HDFS) only provides access control capability at the file level, which could be too coarse-grained for users' expectations. And 2) the heterogeneity of the security models and mechanisms from various data sources may cause incompatible access control features and inconsistent capabilities. For instance, if the user aggregates data from multiple sources that all contain sensitive attributes, the security guarantee is only as good as the weakest link among all data providers.

Security middlewares, such as Apache Sentry [5] and Apache Ranger [4], have been developed to provide fine-grained role-based access control solutions for heterogeneous data sources. However, they do not support access policies that control data usage, such as "user could run statistical functions on sensitive data but could not see the raw data." Unfortunately, such operations are often the purpose of many big data analysis applications, and the corresponding policies are the primary access control intentions of the data owners in big data sharing scenarios [51]. For example, when e-commerce platforms collaborate with retailers and advertisers, they would allow the collaborators to run data analytics algorithms or issue aggregate queries on the transaction database, so that the collaborators may analyze the market, discover sales trends/patterns, and optimize their business strategies. Meanwhile, the e-commerce platforms also need to ensure that the collaborators' queries could never access the raw data, which is sensitive and private.

The purpose-based access control (PBAC) model [32, 37, 38] was proposed for privacy-preserving access control. PBAC was initially designed for conventional RDBMS, in which specific purposes of data usage were defined and translated to a set of SQL queries to be authorized. For example, the purpose of "shipping" allows querying the shipping addresses of active orders [32]. However, in big data analytics applications, it could be difficult to directly associate the abstract-level *data usage purposes* to system-level data processing logic and database operations. For instance, to achieve the data usage purpose "analyzing sales trends", many different algorithms could be employed, including regression analysis, time series analysis, stochastic models, etc. It is almost impossible to specify a static set of database operations to be allowed under this purpose. Meanwhile, the same operation may be a building block in satisfying different data usage purposes. It could be difficult for the database engine to automatically identify the right purpose for an operation, and to allow or deny the operation based on the purpose.

To tackle the challenges, we propose GuardSpark++, a fine-grained access control mechanism for big data sharing and analysis. We first design a purpose-aware access control (PAAC) model, which introduces *data processing purposes* and *data operation purposes* to conventional PBAC. With PAAC, GuardSpark++ could automatically recognize the data processing purposes from data processing logics, and then make access decisions accordingly. Moreover, we enforce the PAAC model in the Catalyst optimizer of Spark [26]. We add an access control enforcement stage between the analysis and optimization stages of Spark's optimization pipeline. The access control stage generates secure logical query plans according to the specified PAAC policies, so that sensitive data objects are only used in data operations that are consistent with the authorized purposes. To demonstrate the effectiveness of GuardSpark++, we evaluate it with five data sources: network streaming, LFS, HDFS, MySQL and Kafka, and four structured data analytics engines: SQL, ML Pipeline, GraphFrame, and Structured Streaming. Last, we evaluate the efficiency of GuardSpark++ using the TPC-DS benchmark [11], and show that GuardSpark++ only introduces 3.97% average overhead on top of the original Spark.

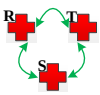
To our best knowledge, GuardSpark++ is the first effort towards a practical purpose-aware access control solution for big data security in Spark. In particular, our main contributions are:

- We have designed a fine-grained purpose-aware access control (PAAC) model with the newly defined data processing purposes and data operation purposes for big data analytics. We further develop analysis algorithm to support automatic purpose recognition, which is the core component for enforcing PAAC. (Section 4)

- We developed a PAAC enforcement mechanism, which provides unified access control support for the heterogeneous data sources (at lower layer) and the higher-layer data analytics engines in Spark.

- We further evaluated GuardSpark++'s effectiveness and efficiency with five data sources (network streaming, LFS, HDFS, MySQL, and Kafka) and four data analytics engines (SQL, ML Pipelines, GraphFrame and Structured Streaming). Experiment results show that GuardSpark++ provides expected security guarantees with a very small computation overhead.

The rest of the paper is organized as follows: we introduce Apache Spark and articulate our motivation in Section 2. We present the threat model and an overview of GuardSpark++ in Section 3, followed by the technical details of the PAAC model and the purpose analysis algorithm in Section 4, and the PAAC enforcement mechanism in Section 5. We then present the experimental results and security analysis in Sections 6 and 7. Finally, we briefly survey the literature in Section 8 and conclude the paper in Section 9.



Example Tables: **(a) doctor table**

Id	Name	Age	Roles	Hospital
1	Bob	28	dermatologist	R
2	Alice	25	neurologist	S
⋮	⋮	⋮	⋮	⋮

(b) patient table

Id	Disease	Expense	PatientName
101	gastric cancer	8000	Aaron
102	cerebroma	9300	Brown
103	neuralgia	4000	Camille
104	dermatitis	2000	Hannah
⋮	⋮	⋮	⋮

An Example Application:

```
patient.selectExpr("PatientName", "Expense as exp1").filter("exp1 > 6000")
.groupBy("PatientName").sum("exp1").select("*").show()
```

Figure 1: A medical data sharing scenario: example tables and an example application based on DataFrame APIs.

2 BACKGROUND AND MOTIVATION

2.1 Introduction to Apache Spark

Apache Spark is a unified computing engine in big data ecosystem, and its layout contains three layers: application layer, optimization layer and execution layer [35].

At application layer, Spark supports structured data analytics engines/APIs based on data types DataFrame and Dataset[35]. An instance of the Dataset is a distributed data collection. A DataFrame is a special Dataset with column style, like a table in relational databases. DataFrame/Dataset can be created on a variety of data sources, such as structured data files in HDFS and RDBMS's tables. 1) The SQL engine in Spark views a DataFrame as a table [26]. 2) MLlib's spark.ml realizes ML Pipelines with a uniform set of high-level APIs built on DataFrame APIs [57]. 3) Based on DataFrame APIs, GraphFrame enhances and extends graph algorithms from GraphX [42, 48]. 4) Structured Streaming [25], an extension from DStream [73], is built on DataFrame APIs.

Building data analysis application is a series of transformations on DataFrame/Dataset — each transformation can be viewed as one step of data operation. A SQL API swallows a data processing fragment (possibly containing multiple steps of transformation) and returns a new DataFrame; a DataFrame/Dataset API swallows a single transformation step and returns a new DataFrame/Dataset. The results in an ultimate DataFrame/Dataset can be obtained by users through invoking an action (e,g., *show* to display data).

To improve efficiency, Spark uses Catalyst [26], a unified logical plan optimizer for structured data analytics engines/APIs, to optimize various data processing logics from applications. In Catalyst, a logical plan includes various data processing logics of an application. Through reforming logic in logical plan, Catalyst generates optimized plan used to produce results [6, 26, 74] at execution layer.

2.2 Motivation

A medical data sharing scenario is shown in Figure 1. We will use it as a running example in this paper.

Example 1. The medical database of *Hospital R* contains a `doctor` table and a `patient` table, as shown in Figure 1. The database is shared with other hospitals in the Regional Health Information Organization (RHIO). Dr. Bob from Hospital R is authorized to access the `patient` table without any restriction. Meanwhile, we would like to specify that Dr. Alice from *Hospital S* is not authorized to see `Disease`, `Expense`, and `PatientMame` columns, but she could run statistical queries and data analytics on those columns.

Note that we currently do not have a widely adopted mechanism to specify or enforce such access control intentions in the conventional access control paradigm. Meanwhile, the big data management and sharing platforms, such as Spark, do not support fine-grained access control mechanisms. From the data owners' perspective, the following access control functionality is expected.

Example 2. A sample data analytics query on the `Patient` table is shown in Figure 1. It calculates the sum of "large spendings" (> 6000) for each patient. According to the access control intentions explained in Example 1, both Alice and Bob should be able to execute this code. Bob's answer shall include two columns (`PatientName` and `sum(exp1)`), while Alice's answer should not contain the `PatientName` column. Meanwhile, the statistical results in `sum(exp1)` should remain identical in Alice's and Bob's answers, since we intend to allow Alice to run statistics on this table.

Intuitively, we would attempt to employ the conventional access control paradigm in the scenario. The Role-Based Access Control (RBAC) model [61] could explicitly prohibit Alice from accessing the `PatientName` column, however, Alice's query would be denied as well, since she uses `PatientName` column in `groupBy`.

The Purpose-Based Access Control (PBAC) model [32] authorizes queries based on pre-identified purposes. However, only abstract-level *data usage purposes*, such as "analysis", "research", or "billing-auditing", are defined in PBAC. In big data applications, each data usage purpose may cover a wide range of queries and data operations, and it is difficult to translate abstract purposes into specific data operations or vice versa. In this example, we cannot specify and enforce that "*Alice may compute with* `PatientName` *in data analytics queries but she cannot include data from this column in the output*". Meanwhile, the conventional PBAC model does not support fine-grained access control at column level, so that all the columns are governed under the same data usage purpose. In particular, when different columns are involved differently in the query (or data processing) logic, PBAC cannot treat them differently.

Example 3. The data processing logics for the two attributes in the sample data analytics query discussed in Example 2 are:
Logic1 (`PatientName`): "patient" → "selectExpr" → "[filter]" → "groupBy" → "[sum]" → "project"
Logic2 (`Expense`): "patient" → "selectExpr" → "filter" → "[groupBy]" → "sum" → "project"
Note that "[operation]" indicates that the attribute is carried in an operation but is not involved in computing. **Logic1** is used to compute "groupBy", while **Logic2** is used to compute "filter" and "sum". Ideally, a fine-grained access control mechanism would handle these logics differently, and also handle purposes of different operations differently. So that Alice could execute Logic1 except the last "project" operation (Alice can compute with `PatientName` but cannot see raw data). She can also compute the entire Logic2 (Alice can read and compute with `Expense`).

Conventional PBAC mechanisms cannot enforce different purposes on different query logics, or specify fine-grained purposes (e.g., read or compute) on different operations. Therefore, the objective of this project is to design and enforce a fine-grained purpose aware access control model that: 1) supports attribute/column-level authorization based on the fine-grained data processing logic for each attribute/column; and 2) supports operation-level authorization based on fine-grained data operation purposes.

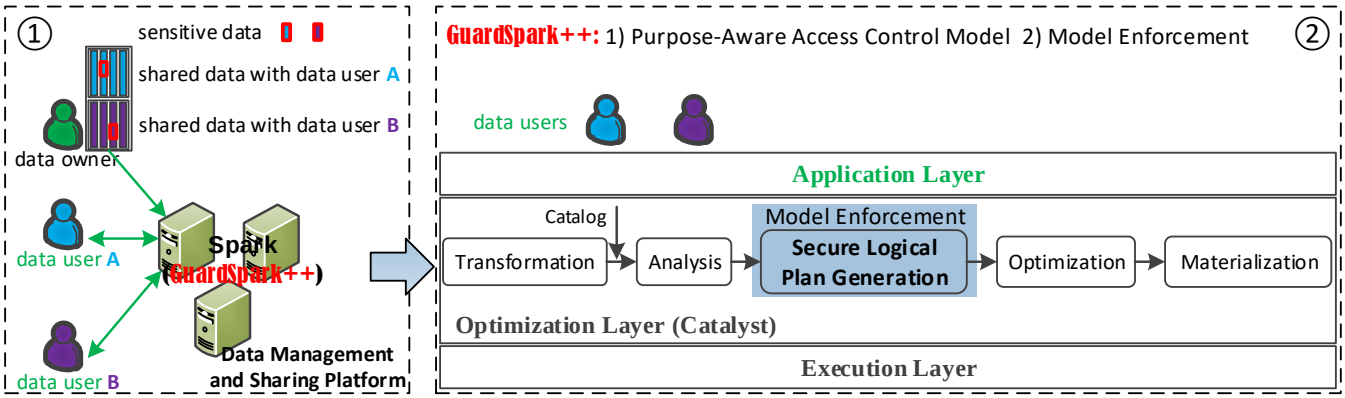


Figure 2: Threat model and solution overview.

3 THREAT MODEL AND SOLUTION OVERVIEW

3.1 Threat Model

The proposed solution contains two main components: the fine-grained, purpose-aware *access control model*, and the *access control enforcement mechanism*. The access control model is suitable in a wide range of big data analytics applications (the generalization of PAAC will be addressed later), while the enforcement mechanism is specifically developed for Spark. The primary stakeholders in the big data sharing and analytics scenario are the *data owners*, the *data management and sharing platform*, and the *data users*.

• The data owners are fully trusted. They place the data on data management platforms, which are owned by them or trusted third parties. Nevertheless, the owners have all the privileges on the data. Some data objects are sensitive that they are only shared with specified users, as shown in Figure 2 ①. The goal of this project is to ensure *data confidentiality* through access control, i.e., only authorized users could perform authorized operations on authorized data objects. The data owners will utilize the access control model supported by the data management and sharing platform to specify access control policies to be enforced by the platform.

• In this project, the data management and sharing platform, including software and hardware, is assumed to be secure and fully trusted by both the data owners and the users. It is expected to faithfully and correctly enforce access control policies.

In real-world applications, the platform may be untrusted and/or compromised so that data confidentiality is jeopardized. For instance, the cloud or the database may not be trusted with plaintext data, system vulnerabilities may cause data/privacy leakage, the implementation of access control enforcement may be buggy, etc. Significant amount of research efforts have been devoted to these topics. Cloud, OS, big data platform and software security issues are considered outside of the scope of this paper.

• From access control perspective, the users are correctly and securely authenticated by the platform. Their roles/attributes are properly managed. They only use the designated data analytics engines or APIs to access data, i.e., they cannot bypass the access control mechanism to directly read from the underlying data sources using python, R, java, scala APIs, or the Resilient Distributed Datasets APIs [72]. Meanwhile, the users are assumed to be curious–if the access control policies or enforcement mechanisms accidentally give them access to an data object that they are not supposed to, they will utilize the access rights that violates the data owners' access control intentions. Last, the users will not attack the software/hardware of the platform (in this paper, Spark).

3.2 Solution Overview

In this paper, we present the GuardSpark++ solution, which contains a purpose-aware access control (PAAC) model and a PAAC enforcement mechanism for Apache Spark.

The PAAC model introduces the concepts of data processing purpose and data operation purpose to the conventional PBAC model. The *data operation purpose* (DOP) defines the purpose of each data operation in data processing logic, e.g., each step in the query logics shown in Example 3. Therefore, a sequence of DOPs are extracted from each data processing logic. The *data processing purpose* of the data processing logic is identified as the most significant DOP within the sequence. A PAAC policy specifies acceptable data processing purposes from a subject (user or application) on a data object. We further develop a purpose analysis algorithm to examine the logical query plans and extract data operation and processing purposes automatically.

Next, we develop a PAAC enforcement mechanism to be hosted in the Catalyst optimizer of Apache Spark. As shown in Figure 2 ②, the original optimization pipeline of Catalyst includes four stages: transformation, analysis, optimization, and materialization. We add a new stage, named *secure logical plan generation*, between the analysis and optimization stages. The added stage enforces PAAC by comparing intended purposes extracted from logical plans against all allowed purposes. In this way, GuardSpark++ transforms analyzed logical plans into secure logical plans that comply with PAAC policies specified by data owners. Subsequently, the secure logical plans are further optimized to eliminate any potential overhead induced by access control. GuardSpark++ also provides pre-defined secure logic templates of secure logical plans for the Structured Streaming engine, which originally utilizes analyzed logical plans as pre-defined logic templates [21, 25].

4 PURPOSE-AWARE ACCESS CONTROL

In this section, we first formally present the fine-grained purpose-ware access control (PAAC) model, and then describe a purpose analysis algorithm, which will be essential in PAAC enforcement.

4.1 The Purpose-Aware Access Control Model

In general, an access control model contains the following core components: {*subject*, *action*, *object*, [*context*], "allow|deny"}. Different access control models specify these component differently, e.g., the ABAC policies combine attributes to specify authorizations.

In the proposed Purpose-Aware Access Control (PAAC) model, the *object* is any data object that could be referenced in a structured data model. In big data processing platforms such as Spark, it could be a data object from any *structured data source* that feeds data to the platform, including 1) tables in relational databases, 2) structured files treated as tables in distributed storage systems (e.g., HDFS) or local file systems, 3) streaming data treated as a table, and 4) other data with explicitly defined column structures. Fine-grained protection of data objects is supported at column-level, row-level, and cell-level. Data owners could define a protected data object using any attribute, e.g., owner or source.

Example 4. Hospital R hosts the medical database shown in Figure 1 in a MySQL DBMS identified as 196.168.12.110:3306. The data object `Expense` attribute of the `patient` table is referred to as *{table:patient, column:Expense, ⋆}*, in which ⋆ denotes *(owner:R and source:MySQL(196.168.12.110:3306:medical))*. We denote this data object as **O** hereafter. In the same way, the second record is referred to as *{table:patient, columns:(Id, Disease, Expense, PatientName), boolean:Id=102, ⋆}*. Finally, Brown's expense is denoted as *{table:patient, column:Expense, boolean:PatientName='Brown', ⋆}*.

Now we formally define the *data processing purpose* and the *data operation purpose*. When a user or an application issues a query or a data analytics algorithm, the query/algorithm will be internally translated to an executable data processing plan (query plan). In practice, the data processing logic is usually organized in a tree structure (i.e., query tree) in the query processor, in which leaf nodes are data objects and internal nodes are operations. A leaf-to-root path represents a data processing logic of the specific data object at the leaf node [1]. The ultimate objective of this data processing logic is the data processing purpose for this data object. Note that a data object may have multiple data processing purposes, when it is involved in different leaf nodes and processed differently in the query plan.

Definition 1 (Data Processing Purpose)**.** The data processing purpose indicates the ultimate purpose of a data processing logic for a data object in a big data application.

While we may ask the user or application to manually indicate a data processing purpose along with the query or data analytics algorithm, however, we cannot trust the self-claimed purpose and use it to enforce access control. In practice, the data processing purpose needs to be discovered at the data management platform based on the query/algorithm content. Meanwhile, we also observe that it is difficult to directly identify the data processing purpose due to the complexity of the query or the algorithm. Therefore, we further decompose the data processing purpose into fine-grained *data operation purposes*.

Definition 2 (Data Operation Purpose)**.** A data operation purpose is the purpose(s) of an operation in the data processing logic of a query or a data analytics algorithm.

Example 5. If we examine the data processing logics introduced in Example 3, the sequences of data operation purposes (DOP) are:
DOP-S1 (`PatientName`): "retrieve" (selectExpr) → "carry" (filter) → "carry/assist" (groupBy) → "carry" (sum) → "output" (project)
DOP-S2 (`Expense`): "retrieve" (selectExpr) → "retrieve/assist" (filter) → "carry" (groupBy) → "compute" (sum) → "output" (project)
Note that we show the data operations, such as selectExpr and filter, together with each purpose. They are *not* part of the purpose. In DOP-S2, `Expense` takes two roles at the "filter" operator: 1) some Expense elements are retrieved from the set and passed to the next operation; and 2) the data object is also used as an operand in the Boolean expression in an assistance role. `PatientName` also takes two roles in the "groupBy" operator: assist and carry.

Data operation purposes are defined by the data management and sharing platforms and provided to the data owners, who are expected to use them to specify access control policies. Meanwhile, the platforms are also expected to automatically recognize the predefined purposes from query plans, so that the access control policies could be enforced accordingly. In this project, we demonstrate the capacity of PAAC using five sample purposes:

Retrieve (DOP-R). When a data object is retrieved from the data source or processed through a selection (filter) function, the corresponding data operation purpose is DOP-R. It is the *de facto* pre-requisite purpose of many other purposes, since data must be retrieved before it can be used.
Compute (DOP-C). When the data object is an operand of a computing operation and it is transformed in this operation, the data operation purpose is DOP-C. In Example 5, data object `Expense` is aggregated in the "sum" operation, hence, the purpose is DOP-C.
Assist (DOP-A). When a data object is involved in an operation but its value is not changed, the data operation purpose is DOP-A, i.e., the data object takes an assistance role in the operation. In Example 5, `PatientName` assisted in the "groupBy" operation.
Carry (DOP-Ca). When a data object is carried during an operation but not otherwise involved, it is denoted as DOP-Ca. In Example 5 [2], `PatientName` is carried in operation *filter("exp1 >600")*, hence, its data operation purpose is DOP-Ca at this step. Note that the "carry" purposes do not impact data operation or access decisions at all, hence, they could be safely ignored in access control enforcement.
Output (DOP-O). When a data object is returned to the user or application, the data operation purpose is DOP-O. In big data applications, whether a DOP-O purpose could be allowed often relies on the previous operations on this data object. For example, "DOP-R-O" may be denied, while "DOP-R-C-O" could be allowed.

[1] We would like to note that a data processing logic in the assistance role may end as an operand in an internal node and not carried to the final output. They are not discussed here but they are properly handled in access control enforcement.

[2] The join operations are not shown in the examples. In practice, each join operation (i.e., natural join or theta join) contains a Boolean condition that indicates which data objects are used to assist in pairing data while other data objects are carried. Thus, one join operation indicates two kinds of operation purposes: DOP-A and DOP-Ca. In addition, the union operation and Cartesian product operation are DOP-Ca.

As we discussed earlier, a leaf-to-root path in the query tree represents a data processing logic of the data object at the leaf. The data operation purposes of this object are sequentially concatenated along the path. Ideally, the data owners may specify all acceptable (or denied) patterns of data operation purposes, such as "`Expense`: DOP-R-*-C-*-O", where * indicates wildcard. Meanwhile, the big data platform is expected to enforce the policies by matching every path of a query tree against the patterns. However, usability is a concern due to the complexity of the patterns–the data owners may not want to define every allowed or denied pattern, or they may be incapable of doing so. In this project, we present a simplified model that captures the *key purpose* in each data operation logic, which shadows all other purposes in the same sequence. In particular, we observed that: 1) All the data operation purposes are sequentially applied to the data object, while only a subset of the operations may modify the data, e.g., DOP-C. 2) The total modification to the data object is no less than the most significant modification in the sequence of operations. 3) From data protection perspective, the data owner would specify how much modification needs to be performed on a data object before it may be sent to output, or if a data object cannot be sent to output. With these observations, we add a new attribute *priority* to each pre-defined data operation purpose, which denotes the level of modifications (i.e., impacts) this operation would add to the data object. In the five data operation purposes we defined in this project, DOP-C has the highest priority, DOP-A and DOP-R has lower priority, while DOP-Ca has NULL priority. Note that DOP-O is a special purpose that will be handled differently than other purposes in the sequence.

Eventually, the data operation purposes in a sequence collectively determine the data processing purpose for the data object. In our simplified model, the data processing purpose is denoted as two-tuple {DPP, DOP-O|NULL}, where the DPP is the highest-priority DOP in the sequence, and "DOP-O|NULL" denotes whether the data object is eventually sent to user after the sequence of operations.

Example 6. Following Example 5, the data processing purpose of DOP-S1 is denoted as DPP1(`PatientName`) {DOP-A, DOP-O}, while the data processing purpose of DOP-S2 is DPP2(`Expense`) {DOP-C, DOP-O}. According to the access control intention introduced in Example 1, DPP2 is allowed to Alice, while DPP1 is denied.

Finally, we define the PAAC access control policy.

Definition 3. An purpose-aware access control policy is a 4-tuple $\langle \mathbb{S}, \mathbb{O}, \mathbb{E}, \mathbb{P} \rangle$, where $\mathbb{S}$ demotes the subject, $\mathbb{O}$ denotes the object, $\mathbb{E}$ specifies the environment conditions that the requests must satisfy, and $\mathbb{P}$ denotes the data processing purposes that are *allowed*.

Note that the PAAC model may employ any existing access control model, such as RBAC or ABAC, to identify the subject (user or application). Last, we give a sample PAAC policy.

Example 7. A PAAC policy for Alice, as introduced in Example 1, may be specified as: ⟨Alice, **O**, **E**, {DOP-C, DOP-O}⟩, where **O** is defined in Example 4, **E** denotes the additional conditions such as Alice's visiting IPs, and the purpose explicitly allows Alice to compute with the data object (`Expense` column) and view the result of the computation. Similarly, we can also specify purpose {DOP-A, NULL} on `PatientName` to specify that Alice could use this column to assist data analytics (e.g., in groupBy), but she cannot include it in the output.

Algorithm 1 Purpose Analysis Algorithm

Require: $\mathbb{DOPIPE}$: data operations pipeline for an object O.
Ensure: $\mathbb{PAG}$: purpose analysis graph, each node containing operation purpose and purpose' priority attributes.

1: $\mathbb{PAG} \leftarrow \emptyset$
2: **for** each OPERA $\in \mathbb{DOPIPE}$ **do**
3: Determine OPERA's purpose(s) according to the operator or function.
4: **if** OPERA has one purpose on O **then**
5: Append one node to each branch in $\mathbb{PAG}$.
6: **else**
7: Append multiple nodes to each branch in $\mathbb{PAG}$.
8: **end if**
9: **end for**
10: **for** each PATH $\in \mathbb{PAG}$ **do**
11: DPP $\leftarrow$ the highest-priority operation purpose of PATH.
12: Bind DPP with PATH.
13: **end for**

4.2 Purpose Analysis Algorithm

Based on the structured data analytics engines/APIs in Spark, users use column-level objects to express various data operations in all data processing logics of an application. Now, we present an algorithm to automatically identify the data operation purpose sequence of each data processing logic.

We view the lifecycle of a data object in an application as a pipeline of data operations. The operations pipeline can be obtained directly — the application codes contain all data operations information, and the logical plan in Spark Catalyst is also the container of data operations. Each object in the pipeline sequentially passes through the data operations. For example, the `PatientName` and `Expense` objects have the same pipeline as shown in Example 3. However, each object is used differently in these operations and has its own sequence of data operation purposes, such as DOP-S1 and DOP-S2 in Example 5.

The purpose extraction algorithm first extracts all data operation purposes (DOPs) from the operations along the pipeline for a data object, and then generates a purpose analysis graph for the object and identifies the data processing purpose (DPP) for each path. The algorithm is shown in **Algorithm** 1.

• The DOP of each operation is determined by the operator or function (Line 3). For example, the "sum" function and its input object "exp1" determine DOP-C purpose on the input object; the "filter" function and its predicate ("exp1 > 6000") determine the DOP-A purpose of the object in the predicate.

• We follow the data operations pipeline to add nodes to the purpose graph (Lines 4-8). Each node in the graph contains an operation purpose and its priority. For an operation with one purpose, we directly append a node to the graph. Meanwhile, a data operation may have multiple purposes, e.g., the data object `Expense` takes two DOPs at the "filter(exp1>6000)" operation, DOP-R and DOP-A. For an operation with multiple purposes, we split the path so that each new DOP is added to a branch. Note that when a branch already has

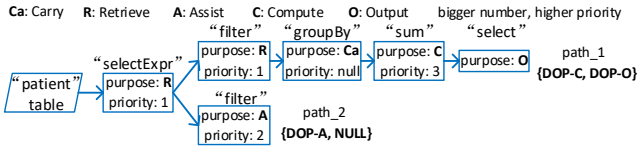


Figure 3: Purpose analysis graph of data object Expense.

the highest-priority DOP, we stop appending nodes to this branch, except for the final DOP-O (if any). In Figure 3, we demonstrate the purpose graph of data object `Expense` in Example 5.

• Finally, we identify the highest-priority DOP in each path of the graph and label it as the DPP of the the path (Lines 10-13). For example, the DPP of path_1 in Figure 3 is DOP-C, which is the purpose of the "sum" operation.

The generated purpose graph captures all DOPs for each data object. Each path in the graph represents a data processing logic and eventually indicates its data processing purpose. For example, the path_1 is the main data processing logic of `Expense`, which indicates its DPP as {DOP-C, DOP-O}.

5 PAAC ENFORCEMENT IN GUARDSPARK++

In this section, we present the details of the design and implementation of the PAAC enforcement mechanism on Spark.

5.1 GuardSpark++ Architecture

We choose to enforce the purpose aware access control model in Catalyst due to the following reasons: 1) All the structured data analytics engines/APIs are built on top of Catalyst, so that queries and data analytics algorithms from these APIs must go through Catalyst. Therefore, building GuardSpark++ in Catalyst ensures that access control is enforced on all the requests from designated sources. 2) The original (before optimization) logical query plans are directly accessible at the Catalyst optimizer, so that we can efficiently and effectively examine, modify or deny the query plans. 3) The modified query plans are further optimized by Catalyst, so that modifications by GuardSpark++ will not affect query performance.

To generate secure logical plan, we design four core modules based on the PAAC model: data object recognition, purpose analysis, compliance checking, and compliance enforcement. The first two modules utilize analyzed logical plan to recognize objects and each object's data operation/processing purposes, and to determine object and each object's purpose. To make access decisions, the third module evaluates subject, object, environment and purpose against access control policies. The forth module enforces access decisions on analyzed logical plan to produce secure logical plan.

Example 8. As shown in Figure 5, the analyzed logical plan of the example code in Section 2.2 is used to exemplify secure plan generation. An *analyzed logical plan*, a tree-like data structure, utilizes ordered data operations on column-level objects to describe each data processing logic in an application [26]. Each node in analyzed logical plan is an operator (e.g., Project, Aggregate and Filter) which encapsulates a set of expressions[3]. The leaf node

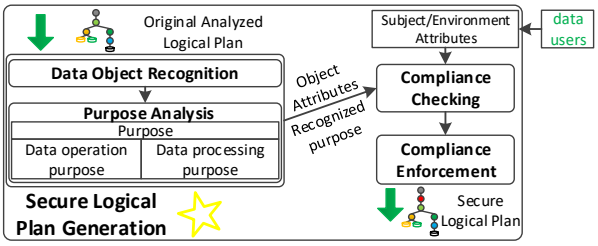


Figure 4: Secure logical plan generation stage

operator encapsulates raw data objects in data source. Each internal operator encapsulates which objects the operator is inputted, how objects are operated, and which objects (after being operated) are delivered to its following operator(s) or are returned to data users.

5.2 Data Object Recognition

The column-level objects in analyzed logical plan are recognized. This does not mean that GuardSpark++ cannot control the access to row-level and cell-level objects. The module aims to recognize raw and alias column-level objects in plan and describe those objects using object attributes — each raw object is analyzed data operation/processing purposes by purpose analysis module (Section 5.3) (relevant alias objects participate); raw object attributes and its purpose(s) are used by compliance checking module (Section 5.4).

All raw objects are recognized from leaf node operators in plan. For example, the `Expense` column of the "Relation" operator in Figure 5 is a raw object, and can be described as *{owner:R, source:MySQL (196.168.12.110:3306:medical), table:patient, column:Expense}*.

However, besides the raw objects, the alias mechanism [4] in Spark produces alias objects in order to facilitate the description of data processing logic in the plan [35]. Alias objects are contained in non-leaf node operators of the plan (e.g., the first "Project" operator in Figure 5 contains `exp1` alias) and also need to be recognized to accurately analyze data processing purposes (Section 5.3). Because aliases are produced by the alias mechanism of Spark, the owner and source attributes of an alias are null and the table attribute is the operator initiating the alias through an `Alias` expression. For example, `sum(exp1)` object in Figure 5 is described as *{owner:null, source:null, table:Aggregate, column:sum(exp1)}*. Because aliases are located in non-leaf nodes of plan, GuardSpark++ must traverse plan; as sub-products, a series of operators passed by each object are obtained and form an operator-processing path. As shown in Figure 6, these three objects: `Expense`, `exp1`, `sum(exp1)` have three operator-processing paths. Lineage exists among paths of each raw object and its relevant alias objects due to the alias mechanism. This lineage is convenient to purpose analysis in Section 5.3.

5.3 Purpose Analysis

This module recognizes data processing purposes for all raw objects in the logical plan. According to the purpose analysis algorithm in

[3]Catalyst has its own expression system. Expression is used to evaluate a result value according to given input values [13, 26]. For example, `Computation` expression (e.g., `SUM` expression, `AVG` expression) can evaluate a computation result of given input values; `Alias` expression is used to evaluate an alias for input expression; `Attribute` expression is able to evaluate an attribute according to input name string; `Predicate` expression is used to evaluate boolean value.

[4]Aliases for any column can be named by users (e.g., in Figure 5, users name the alias of `Expense` as the `exp1`). Many temporary aliases indicating the intermediate results of application are usually produced by Catalyst (e.g., the `sum(exp1)` in the second "Project" operator is this kind of alias). An alias is carried by an `Alias` expression.

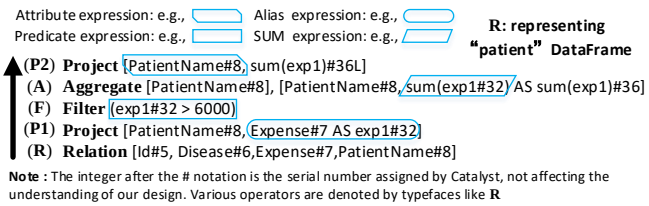


Figure 5: Analyzed logical plan.

Section 4.2, this module first obtains the data operations pipeline for each raw object. Then, it constructs the purpose analysis graph by recognizing data operation purposes along the pipeline. Finally, to identify the data processing purpose, this module finds the highest-priority DOP in each path of the purpose analysis graph. Here we mainly focus on the logical-plan-based method to obtain data operations pipeline and to recognize data operation purpose.

To obtain the data operations pipeline for a raw object in the logical plan, this module finds the complete sequence of operators for the object. The operators are obtained according to the lineage mentioned in Section 5.2. For example, Path ② in Figure 6 is derived from Path ①, while Path ③ is developed from ②. All three paths combined provide the complete sequence of operators of `Expense`.

To construct the purpose analysis graph, this module recognizes each DOP along the sequence of operators for each raw object. In the logical plan, a DOP is identified by the expression encapsulated in the operator node. We employ pre-defined heuristics to recognize operation purpose: 1) DOP-R is assigned to operators which explicitly retrieve data out of leaf nodes or implicitly retrieve object using filter (or similar) functions (e.g., the first "Project" operator gets `PatientName` from `Relation`, and the "Filter" operator implicitly allows `PatientName` and `exp1` to pass). 2) DOP-C is assigned to operators which make computation on data objects (e.g., in Figure 5, the DOP-C on `exp1` is assigned to the "Aggregate" operator which makes computation on the object using `SUM` expression.). 3) DOP-A is designated by operators which use column-level object as assistance in an expression (e.g., the "Filter" operator uses `exp1` in the expression "exp1#32 > 6000" to filter data.). Last, if the sequence of operators ends at the root of the logical plan, DOP-O is recognized. Note that we ignore DOP-Ca purposes as explained in Section 4.1.

5.4 Compliance Checking

This module makes access decisions by evaluating identified objects, subjects, environments and purposes against policy rules. Here we explain how the recognized purposes are compared with data-owners-specified data processing purposes of column-level, row-level or cell-level objects (Section 4). Meanwhile, there are existing works in the literature to handle the subject and environment.

GuardSpark++ evaluates each recognized column-level object as follows: We first identify all relevant fine-grained access control policies using subject and object attributes. Then, we employ the following heuristics to identify the allowed cells for each kind of data processing purpose on the column. For the object, we can directly extract data processing purposes from each purpose-bound path in its purpose analysis graph. 1) If the purpose of a path is {DOP-R, DOP-O}, we find retrieval-regulating rules from selected rules in the second step, and bind allowed cells with the path. 2) If the purpose of a path is {DOP-C, DOP-O}, we find computation-regulating rules from selected rules in the second step, and bind allowed cells with the path. 3) If the purpose of a path is {DOP-A, NULL}, we find assistance-regulating rules from selected rules in the second step, and bind allowed cells with the path. As a result, each access decision means allowed cells of each data processing purpose in a purpose analysis graph.

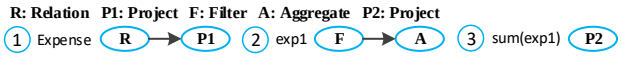


Figure 6: Operator-processing path examples.

5.5 Compliance Enforcement

With the access decisions from Section 5.4, GuardSpark++ transforms the original logical plan into a secure query plan using query rewriting — modifying operators or producing guard operators to insert secure operators at appropriate enforcement positions in the original plan.

When the allowed cells are empty, we consider two situations. 1) If data processing purpose is {DOP-R, DOP-O} or {DOP-C, DOP-O}, the enforcement position is the root operator in the original plan and GuardSpark++ uses *zero setting logic* based on the `Alias` expression to replace corresponding expression. For example, if a doctor is prohibited from using DOP-C on the expenses, the `sum(exp1)` will be respectively replaced with "0 AS sum(exp1)". This replacement obviously has no effect on other processing purposes, and obeys the immutable schema structure of immutable DataFrame/Dataset [6, 26]. 2) If {DOP-A, NULL}, the enforcement position is operators connected with the assistance purpose and GuardSpark++ deletes all expressions relevant with the assistance purpose. For example, if a doctor is not allowed to use `Expense` for assistance purpose, the expression in "Filter" operator is deleted. In the above cases, we insert secure operator through modifying an operator.

When the allowed cells are non-empty, a secure expression is constructed to retain them. A guard operator uses the secure expression to retain allowed cells and erase prohibited cells. A guard operator is described as Guard("allowed_sells", "column") where "allowed_cells" is represented by secure expression and "column" is the targeted column. The enforcement position is below the last operator passed by the corresponding object. For instance, if a doctor is prohibited from counting the expense of two patients "Aaron" and "Brown", a new guard operator "Guard(PatientName#8 != (Aaron OR Brown), Expense)" is inserted below the first "Project" operator.

6 EXPERIMENTAL RESULTS

With Spark as the baseline, we test GuardSpark++'s overhead and scalability. We also evaluate the recognition of data operation/processing purposes in GuardSpark++ through case studies on five data sources and four structured data analytics engines.

6.1 Settings

Hardware and Software Configurations. We conduct our experiments on a cluster of 7 nodes, including one primary and six secondary nodes. Each node is equipped with 32 Intel Xeon CPUs E5-2630 v3 @ 2.40GHz, 130GB of RAM and 4TB of disk capacity,

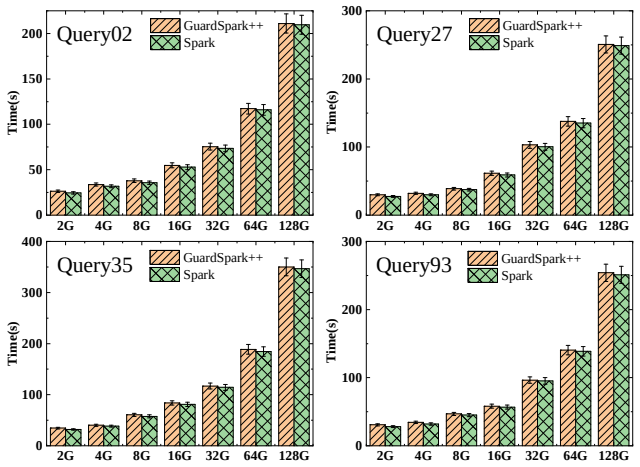


Figure 7: Query processing efficiency of GuardSpark++.

running 64-bit CentOS. HDFS (v. 2.6.0) is used to store the data with a replication factor of 2. We build GuardSpark++ on Spark v. 2.4.0.

Datasets and Benchmarks. To test GuardSpark++'s efficiency and scalability, we use the TPC-DS benchmark [23], which covers various query types in decision support systems and is used as the performance testing framework for Spark SQL in Spark 2.2+ [1, 11]. In our case studies, we also test access control on SQL engine with the TPC-DS benchmark. We use the popular Iris [9] and BigDataBench [69] for the ML Pipelines engine. We use the Pokec [8] for the GraphFrame engine. The datasets used in the tests are stored on HDFS. For Structured Streaming engine, we evaluate network streaming, LFS, HDFS and Kafka sources, and test a common logs analysis system. Finally, we evaluate GuardSpark++ on a relational data source MySQL.

6.2 Efficiency of GuardSpark++

We evaluate the efficiency of GuardSpark++ using 2GB, 4GB, 8GB, 16GB, 32GB, 64GB and 128GB retail datasets generated by TPC-DS data generator. We choose Query02, Query27, Query35, and Query93 in TPC-DS package as they cover various data operations and contain complex data processing logics. In particular, Query93, Query27, and Query02 have minimum, moderate and maximum number of expressions designating operation purposes, respectively, while Query35 contains three sub queries. The queries contain the following data processing purposes: {DOP-R, DOP-O}, {DOP-C, DOP-O} and {DOP-A, NULL}. With Spark as the baseline, to ensure dataset size consistency, we customize access control policies in GuardSpark++ in terms of queries' own operation information. For example, if the condition in WHERE clause of a query means the value of column 5 in a relation equals to 2 and the column is retrieved and outputted, we can state the policy that its value for {DOP-R, DOP-O} should be greater than 0. Table 1 shows all policies of the four queries. In this way, we can avoid data size change because of access control, and only probe into the cost of our control. Each query is repeated 100 times on each dataset, and the average execution time is calculated as the final index.

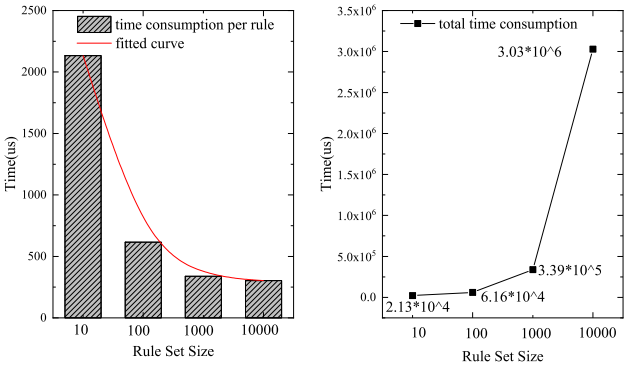


Figure 8: The scalability of GuardSpark++ against access control policy size.

As shown in Figure 7, GuardSpark++ introduces a small overhead to Spark: 8.44% (2GB dataset), 6.40% (4GB), 4.69% (8GB), 3.52% (16GB), 2.17% (32GB), 1.65% (64GB) and 0.89% (128GB). Apparently, the relative overhead is below 10% in all cases and gradually decreases with larger datasets. This is because Spark's query execution time increases with larger datasets, while the time to produce a secure logical plan stays stable. The result shows that GuardSpark++ is highly scalable, which could be used in big data sharing scenarios, where exabytes or even zettabytes of data may be evaluated.

6.3 Policy Scalability

The data scalability of GuardSpark++ is discussed in Section 6.2. Next, we measure the scalability of GuardSpark++ with respect to the size of the access control policy. According to the architecture of GuardSpark++, the policy scalability is mainly determined by the purpose compliance checking module. To directly evaluate how this module affects the scalability of GuardSpark++, we conduct experiments with the following access control rule set sizes: 10, 100, 1000, and 10000. For each rule set, we execute Query02 100 times and compute the average query processing time in the purpose compliance checking module.

Figure 8 (left) shows the average purpose compliance checking time execution per-rule. As shown in the figure, the per-rule execution time reduces significantly when the rule set size grows from 10 to 100. Figure 8 (right) shows the growth of the total compliance checking time when the rule set size grows. Note that the X-axis is in logarithmic scale. As shown, when the size of the rule set size increases, the total compliance checking time increases approximately linearly. The total compliance checking time for 10000 rules is approximately 3 seconds, which implies that the proposed mechanism is highly practical.

6.4 Case Study

Next, we show some use cases from four data analytics engines and five data sources. The following empirical results show that GuardSpark++ works effectively and correctly.

6.4.1 SQL. We perform the assessment on the TPC-DS benchmark which contains various tables. These tables record much sensitive information. For instance, the "customer_address" table contains addresses of all customers; the "customer_demographics" table records demographics of all customers; the "store" table includes

Table 1: Access control policies specified in GuardSpark++ for efficiency test

All data objects in these queries, except the column objects in policies, are used without restriction.
We only show protected data and its specified data processing purpose in policy.

Query	Protected Object and Specified Purpose
Query02	the value (>0) of column 5 in table data_time is allowed {DOP-R, DOP-O}.
Query27	the value (>0) of column 13 in table store_sales is allowed {DOP-C, DOP-O}.
Query35	the value (!='no') of column 2 in table customer_demographics is allowed {DOP-R, DOP-O}.
Query93	the value (!='-1') of column 4 in table store_sales is allowed {DOP-R, DOP-O}.

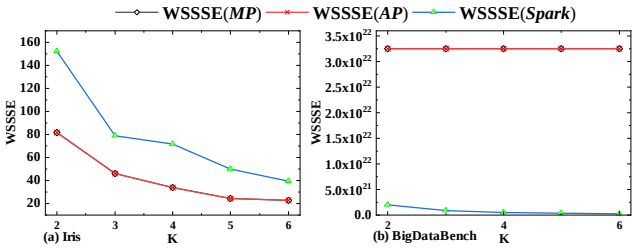


Figure 9: Using Iris dataset and BigDataBench to measure K-WSSSEs. We run GuardSpark++ to automatically enforce access control policy (*AP*) and simulate the policy enforcement by manually deleting corresponding data (*MP*); we run Spark without policy enforcement (*Spark*).

addresses of all stores; the "store_sales" table has sale price of each commodity. The access control policies protect all sensitive information from being seen.

To show the assessment, we select the Query35 (Figure 10 in Appendix A) in the TPC-DS benchmark. This query discloses consumption dependent and some information about living address, gender and marital status [23]. This query, which contains various data processing purposes, aliases and sub queries, is complicated enough for showing our assessment. The assessments on other queries are announced on the website (https://github.com/liveonearthormars/SparkSQL-test).

The Figure 11 in Appendix A shows detailed analysis about the control on data processing purposes. Particularly, we utilize zero setting logic (Section 5.5) to prevent sensitive data from being seen while various computation results are correct. For intuitively demonstrating the control effect, we provide three result sets about Query35 in Figure 12 in Appendix A. Specifically, the computation results in Figure 12(b) are consistent with those in Figure 12(c) because only outputted sensitive information is set to zero according to access control policies. However, the computation results in Figure 12(a) are different from those in both Figure 12(b) and Figure 12(c) because directly deleting sensitive information from data source alters computation results that data users expect.

6.4.2 ML Pipelines. We select a clustering algorithm K-means [17] and regard the Within Set Sum of Squared Error (WSSSE) as the metric. First, we use Iris dataset [9], and access control policy regulates that the computation of the first column's value less than or equal to 5.5 is prohibited, other columns are used without restriction. We run GuardSpark++ to automatically enforce the policy and obtain the WSSSE set in the case of different K values (2,3,4,5,6). We simulate the policy enforcement by manually deleting the corresponding data from Iris dataset and obtain the WSSSE set based on the same K values. Also, basing the same K values on, we run Spark without policy enforcement to obtain the WSSSE set as a baseline. The experimental results are shown in Figure 9(a). Apparently, the last K-WSSSE set is different from the first two identical K-WSSSE sets. Second, we utilize 4GB data generated using BigDataBench [69]. With the same experimental method above, we obtain the results (similar to the first test) shown in Figure 9(b). Obviously, GuardSpark++ can constrain data usage for ML algorithm. Access control for ML engine is vital when data owners share their data and customize their policies to protect their sensitive data from being used by ML algorithms that may indirectly expose more privacy information about data owners [24].

6.4.3 GraphFrame. We select PageRank algorithm [14] originally used to measure the relative importance of website pages, and set the maximum iteration to 100. We use Pokec social network dataset, each node containing user data about gender, age, hobbies, interest, education etc [8]. In the dataset, the most influential node has ID 5935 and the PageRank value is 625.1821405718983. We assess the real scenario — data owners share their data containing sensitive data to social network. To protect sensitive data of the most influential node, data owners customize the access control policy that the computation of the second column's value equal to 5935 is banned. As a result, in GuardSpark++, we obtain the most influential node, ID 5876 with PageRank value 287.1705351204417. Obviously, the original node ID 5935 has been isolated. Access control for graph engine is necessary when data owners share their data and customize policies to prevent sensitive data from being analyzed by graph algorithms which may dig out more privacy information about data owners [31, 56].

6.4.4 Structured Streaming. To showcase continuous access control enforcement for Structured Streaming, we select several data sources. First, we test network stream source in the common use case given in [21]. The Netcat (a small utility in Linux) serves as streaming data server; the following sensitive words groups — ("Fund", "Association") and ("Fund", "Association", "Insurance") — are respectively sent by port 9999 of localhost; the same words are counted. We run the use case on GuardSpark++ with access control policy: the sensitive word "Fund" from port 9999 of localhost cannot be computed by user. Each experimental result set does not contain "Fund" after each sending. Second, we leverage file sources (LFS and HDFS [30]) and Kafka source [54] to conduct similar tests to the first, and experimental results show that we still can control which sensitive word cannot be computed. Finally, we construct a

common logs analysis system, which consists of Sysdig (generating logs by its default format) [22], flume (collecting logs) [50] and Kafka (transporting logs to GuardSpark++) [54]. We consider the IP in logs as sensitive data and forbid IP from being seen. As a result, IP information cannot be seen in GuardSpark++.

6.4.5 Various data sources. The previous case studies embody these low-level data sources: LFS, HDFS, network streaming and Kafka — the low-level data read permission in these sources [7, 16, 58, 65] is opened. Next, we select MySQL to measure the effectiveness on relational data source. We use real world data from business domain. Similar to [37], we consider MyCompany which provides catering for elderly people (customers). For better services, MyCompany collects customers' data, including customer table, address table and order table (the customer's "id" is the primary key for each table). MyCompany puts those tables into MySQL and authorizes our Spark cluster to access those tables according to the access control mechanism in MySQL [18]. However, MyCompany customizes the access control policy in GuardSpark++: the "id" attribute value must be greater than 10 when users retrieve and output data. In such case, we use the JDBC APIs in DataFrame to retrieve those tables. The result set only contains records whose ids are greater than 10.

7 SECURITY ANALYSIS AND DISCUSSIONS

The security of GuardSpark++ relies on the following factors: 1) GuardSpark++ correctly decomposes the logical plans and recognizes the data processing purposes, 2) GuardSpark++ correctly enforces access policies that are consistent with the data owner's access intentions, and 3) all user queries that need access control are processed through Catalyst and GuardSpark++.

The correctness of the first two factors is ensured in the design of GuardSpark++. In particular, GuardSpark++ recognizes the DOPs (e.g., DOP-R, DOP-C, DOP-A) and the data processing purpose (DPP) for each data object from its logical plan (Sections 5.2 & 5.3). According to the restriction of the PAAC policies specified by the data owners, GuardSpark++ identifies which cell-level data objects are allowed for which purposes (Section 5.4). With these decisions, the secure logical plan is generated (Section 5.5), which eliminates any unauthorized access to sensitive data at cell-level.

The third factor requires all user queries that are subject to access control to be submitted through Spark's structured data analytics engines/APIs. To enforce this, we can configure Spark to prevent certain users from using the unprotected APIs. Meanwhile, if this requirement is not enforced, a user can read data directly from the underlying data sources in Spark by employing python, R, java, scala APIs, or RDDs APIs. In such case, GuardSpark++ treats user's codes as untrusted codes, and can exploit existing works [46, 47, 66] to circumscribe untrusted codes to immune itself. We would like to note that RDD-based access control enforcement is independent of GuardSpark++ and is an interesting future direction.

Inference Attacks. The inference attack is usually considered a problem beyond the conventional access control paradigm [62, 71]. In GuardSpark++, sensitive data may be subject to inference attacks when 1) the "computing" operations do not effectively fuse or transform information from raw data, such as the "sum" of only one (or a few) value(s); or 2) multiple overlapping or relevant queries are submitted and the returned results are examined. The first attack could be mitigated in a combination of static and dynamic approaches. In static analysis, certain (simple) operations may be excluded from DOP-C purposes, and considered as DOP-R instead. Meanwhile, we could generate a runtime validator, *CompGuard*, in the logical plan for each DOP-C operation. The CompGuard is a Boolean expression that validates the corresponding operation based on its runtime data size. All the CompGuards are aggregated to determine if the query should be allowed or denied during the runtime check. On the other hand, the second attack could be partially mitigated by tuning PAAC policies to deny certain attributes from being used in DOP-A. For example, Alice can use two queries: "patient.selectExpr("Expense").sum("Expense").show" and "patient.filter("PatientName != Aaron").selectExpr("Expense").sum ("Expense").show" to infer Aaron's expenses. However, the attack will be ineffective if the PAAC policy prohibits `PatientName` from the DOP-A purpose. Last, we would like to emphasize that the inference attack is not the focus of this paper. A complete mitigation of inference attacks require more efforts that are beyond conventional access control mechanisms.

Generalization. First, the proposed PAAC model has enough extensibility in response to new access control needs. The data operation purposes introduced in Section 4.1 could be easily extended to handle complex access control intentions. For instance, we can divide DOP-C into "count" and "sum" purposes to allow more specific control of operations and potentially mitigate complicated inference attacks [45, 62, 71]. Meanwhile, the design of GuardSpark++ in Catalyst is a practical example that could be migrated into other big data platforms inspired by SQL, e.g., Presto [19], Spanner [27], Hive [67], SCOPE [76]. With reasonable modifications, the implementation of GuardSpark++ could be adapted into the logical query optimizer or processor of these platforms.

Compatibility with RBAC and ABAC. First, PAAC could employ existing access control models, especially RBAC and ABAC, to specify the subjects (users or applications). PAAC could also adopt other concepts from existing access control paradigms that do not interfere with the concept of purposes, such as resource attributes, contexts, etc. Second, our model may be adapted into the ABAC model [52] by adding purposes into the *environmental attributes* of ABAC. However, as we have explained in the paper, purpose recognition and processing algorithms need to be developed in order to enforce the purposes.

Remaining Attack Surfaces. GuardSpark++ is an access control mechanism, not a silver bullet that addresses all attacks. Conventional attacks, such as DoS [55], collusion attacks [77], or traffic analysis [75], will still work even if GuardSpark++ is in place. In particular, if the Spark platform is untrusted or compromised, GuardSpark++ becomes ineffective. Our research on access control is independent of and complementary to research efforts on protecting big data against the attacks mentioned above.

Future Improvements. The policies in GuardSpark++ can be improved to support more features in addition to the high level DOP-* criteria. 1) We may introduce the obligation concept [36, 59], which can regulate the actions that big data sharing system should take

to allow data owners to have the right to know what is happening to their sensitive data [34, 52]. For instance, to receive the information about all usage purposes in each query statement of any subject, data owner can describe the *obligation* as *{subject:anyone, algorithm:query, purposes:(DOP-R, DOP-C, DOP-A, DOP-O)}*. Based on the obligations, GuardSpark++ can send corresponding information to Spark-external obligation severs which fulfill system's obligation to data owners. In this way, data owners can get more control on their sensitive data. For example, a data owner can prohibit all usage purposes on their sensitive data when he/she is alerted that the sensitive data may have been unlawfully accessed by a user, and then initiates an audit for this user. Meanwhile, because of the supervisory role, data owners have deterrent effect on users. 2) Data users may need to know what happens to their queries/algorithms in GuardSpark++ after they submit them to system for execution. To support this requirement, we could allow data owners to specify policies with corresponding components to identify if the system could provide transparency to the users, and let GuardSpark++ enforces it. However, such transparency to the data users may potentially give adversarial users more information to launch inference attacks.

8 RELATED WORK

In this section, we briefly summarize the literature on access control techniques.

Access Control for Spark. SparkXS [63] is a customized attribute-based access control solution for Spark Streaming. This ABAC solution cannot support purpose-aware access control. A cryptography-based solution [64] targets to protect sensitive data in RDD [72]; the solution does not aim to provide access control solution for protecting sensitive data in big data sharing scenarios. A recently proposed method [15] supports Spark SQL by depending on existing Apache Range policies defined for Apache Hive. Another solution [12] proposed by Databricks enables table-based access control for Spark SQL. The two solutions are not only engine-specific but also do not support purpose-aware access control.

Access Control for Hadoop. Apache Sentry [5], a middleware system, can be deployed between Hadoop runtime engine and data sources, e.g., Hive, Impala, Solr or HDFS, and implements fine-grained role-based access control for Hadoop ecosystem [3]. Although this solution can be unified and fine-grained, it needs to develop new plugins for new data sources and GuardSpark++ does not. Apache Ranger [4] is similar to Apache Sentry. GuradMR [68] provides a fine-grained access control solution for unstructured data in MapReduce system. HeABAC [49] allows the access isolation of collected data in multi-tenant Hadoop ecosystem through ABAC mechanism. None of them provide purpose-aware access control for big data sharing scenarios.

Access Control for NoSQL A recent work [53] improves the access control capability of HBase by customizing fine-grained ABAC for HBase, supporting context-based access control. Depending on SQL++ [60], a unified fine-grained ABAC solution [39] is proposed to improve data security in NoSQL datastores. The two works are not for big data sharing scenarios.

9 CONCLUSION

In this paper, we have proposed GuardSpark++, a novel fine-grained purpose-aware access control model and enforcement mechanism for big data sharing. We first introduce the purpose-aware access control model. In particular, we have defined the data processing purpose and data operation purpose; and then introduced an algorithm to automatically analyze and extract purposes from logical query plans. On the other hand, we have developed a PAAC enforcement mechanism for Spark's structured data analytics engines/APIs. The mechanism is deployed in Spark Catalyst, to re-write logical plans into secure ones on-the-fly. With extensive experiments, we show that GuardSpark++ effectively enforces PAAC and it only introduces a small overhead to Spark. GuardSpark++ is open sourced at (https://github.com/liveonearthormars).

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments. We also thank Dr. Fangxiao Ning's technical help. Tao Xue, Yu Wen, et al. were supported by the Strategic Priority Research Program of CAS, Grant No.XDC02010300. Bo Luo was sponsored in part by NSF CNS-1422206, DGE-1565570, NSA Science of Security Initiative H98230-18-D-0009, and the Ripple University Blockchain Research Initiative. Yingjiu Li was also supported in part by the Ripple University Blockchain Research Initiative.

REFERENCES

[1] AMPlab.University of California, Berkeley.Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.
[2] Apache Beam. https://beam.apache.org/.
[3] Apache Hadoop. http://hadoop.apache.org/.
[4] Apache Ranger. https://hortonworks.com/apache/ranger/.
[5] Apache Sentry. https://sentry.apache.org/.
[6] Apache Spark. https://spark.apache.org/.
[7] The Big Data Security Gap: Protecting the Hadoop Cluster, White Paper, Zittaset, 2014. http://www.zettaset.com/wp-content/uploads/2014/04/zettaset_wp_security_0413.pdf.
[8] By Jure Leskovec. http://snap.stanford.edu/data/soc-Pokec.html.
[9] C.L. Blake and C.J. Merz (1998). UCI Repository of Machine Learning Databases. University of California. http://archive.ics.uci.edu/ml/datasets/Iris.
[10] Data Sharing and Data Integration. https://www.european-big-data-value-forum.eu/data-sharing-and-data-integration/.
[11] Databricks, Spark SQL Performance Tests, 2019. https://github.com/databricks/spark-sql-perf.
[12] Enable Table Access Control. 2017. https://docs.databricks.com/administration-guide/admin-settings/table-acls/table-acl.html.
[13] GitBook, 2019. https://jaceklaskowski.gitbooks.io/mastering-spark-sql/.
[14] GraphFrames User Guide, 2018. https://graphframes.github.io/user-guide.html.
[15] Introducing Row/Column Level Access Control for Apache Spark, 2017. https://ko.hortonworks.com/blog/row-column-level-control-apache-spark/.
[16] Kafka Security, 2019. http://www.cs.toronto-.edu/~kriz/cifar.html.
[17] MLlib:Main Guid, 2018. http://spark.apache.org/docs/latest/ml-clustering.html.
[18] MySQLTutorial. 2019. https://www.mysqltutorial.org/mysql-adminsitration/, 2019.
[19] Presto. https://prestodb.github.io/.
[20] Sharing in the Era of Big Bata. https://sciencenode.org/feature/improving-sharing-in-the-era-of-big-data.php.
[21] Structured Streaming Programming Guide, 2018. http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.
[22] Sysdig. https://sysdig.com/.
[23] TPC BENCHMARK DS Standard Specificationversion 2.3.0, Transaction Processing Performance Council (TPC), 2016. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.3.0.pdf.
[24] Privacy-Preserving Machine Learning: Threats and Solutions. *IEEE Security & Privacy, S&P'19* (2019).
[25] Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18.*

[26] Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data, SIGMOD'15*.

[27] Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD'17*.

[28] Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC'10)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/1807128.1807148

[29] *Big Data Security and Privacy, 2018*.

[30] HDFS Architecture Guide. *Hadoop Apache Project* (2008).

[31] Privacy-Preserving Graph Algorithms in the Semi-Honest Model. In *International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT'05*.

[32] Purpose Based Access Control for Privacy Protection in Relational Database Systems. *The International Journal on Very Large Data Bases, VLDB Journal'08* (2008).

[33] Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[34] The Importance of ABAC: Attribute-Based Access Control to Big Data: Privacy and Context. *Privacy and Big Data Institute, Ryerson University, Toronto, Canada* (2015).

[35] *Spark: the Definitive Guide: Big Data Processing Made Simple, 2018*.

[36] Enforcing Obligations within RelationalDatabase Management Systems. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (2013), 318–331.

[37] Enforcement of Purpose Based Access Control within Relational Database Management Systems. *IEEE Transactions on Knowledge and Data Engineering, TKDE'14* (2014).

[38] Efficient Enforcement of Action-Aware Purpose-Based Access Control within Relational Database Management Systems. In *IEEE International Conference on Data Engineering, ICDE'16*.

[39] Towards a Unifying Attribute Based Access Control Approach for NoSQL Datastores. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 709–720.

[40] *Managing and Sharing Research Data: a Guide to Good Practice*. SAGE Publications Limited.

[41] Privacy by Design in Big Data: an Overview of Privacy Enhancing Technologies in the Era of Big Data Analytics. *arXiv preprint arXiv:1512.06000* (2015).

[42] GraphFrames: An Integrated API for Mixing Graph and Relational Queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, GRADES'16*.

[43] MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM, 2008* (2008).

[44] Big Data Sharing and Analysis to Advance Research in Post-Traumatic Epilepsy. *Neurobiology of disease* 123 (2019), 127–136.

[45] The Inference Problem: A Survey. *ACM SIGKDD Explorations Newsletter* (2002).

[46] A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*.

[47] Going beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2.. In *USENIX Symposium on Internet Technologies and Systems*.

[48] GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14*.

[49] An Attribute-Based Access Control Model for Secure Big Data Processing in Hadoop Ecosystem. In *Proceedings of the Third ACM Workshop on Attribute-Based Access Control (ABAC'18)*.

[50] *Apache Flume: Distributed Log Collection for Hadoop, 2013*.

[51] *Big Data: Storage, Sharing, and Security*. CRC Press, 2016.

[52] Guide to Attribute Based Access Control (ABAC) Definition and Considerations (draft). *NIST special publication* 800, 162 (2013).

[53] An Attribute-Based Fine-Grained Access Control Mechanism for HBase. In *International Conference on Database and Expert Systems Applications*. Springer, 44–59.

[54] Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of the NetDB, 2011*.

[55] Distributed Denial of Service Attacks. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, Vol. 3. IEEE, 2275–2280.

[56] Graph-Based Privacy-Preserving Data Publication. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, INFOCOM'16*.

[57] MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* (2016).

[58] Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security Symposium, USENIX Security'02*.

[59] An obligation model bridging access control policies and privacy policies.. In *SACMAT*, Vol. 8. 133–142.

[60] The SQL++ Semi-Structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL databases. *CoRR* (2014).

[61] Role-Based Access Control. *Network Security Technology & Application* (2007).

[62] *Security in Computing*. Prentice Hall Professional Technical Reference, 2002.

[63] SparkXS: Efficient Access Control for Intelligent and Large-Scale Streaming Data Applications. In *International Conference on Intelligent Environments, 2015*.

[64] Data-at-Rest Security for Spark. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 1464–1473.

[65] Big Data Security: The Evolution of Hadoop's Security Model, 2013. https://www.infoq.com/articles/HadoopSecurityModel/.

[66] Bringing Java's Wild Native World under Control. *ACM Transactions on Information and System Security (TISSEC)* (2013).

[67] Hive-A Petabyte Scale Data Warehouse using Hadoop. In *2010 IEEE 26th international conference on data engineering, ICDE'10*.

[68] GuardMR: Fine-Grained Security Policy Enforcement for MapReduce Systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15*.

[69] Bigdatabench: A Big Data Benchmark Suite from Internet Services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture, HPCA'14*.

[70] MeDShare: Trust-Less Medical Data Sharing among Cloud Service Providers via Blockchain. *IEEE Access* 5 (2017), 14757–14767.

[71] Data Level Inference Detection in Database Systems. In *Proceedings. 11th IEEE Computer Security Foundations Workshop, CSFW'98*.

[72] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

[73] Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP'13*.

[74] Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* (2016).

[75] Statistical Privacy for Streaming Traffic. In *NDSS*.

[76] SCOPE: Parallel Databases Meet MapReduce. *The International Journal on Very Large Data Bases, VLDB Journal'12* (2012).

[77] A Secure Anti-Collusion Data Sharing Scheme for Dynamic Groups in the Cloud. *IEEE Transactions on parallel and distributed systems* 27, 1 (2015), 40–50.

A EXPERIMENTAL TABLE AND FIGURES

```
SELECT ca_state, cd_gender, cd_marital_status, count(*) cnt1, min(cd_dep_count),
max(cd_dep_count), avg(cd_dep_count), cd_dep_employed_count, count(*) cnt2,
min(cd_dep_employed_count), max(cd_dep_employed_count),
avg(cd_dep_employed_count), cd_dep_college_count, count(*) cnt3,
min(cd_dep_college_count), max(cd_dep_college_count), avg(cd_dep_college_count)
FROM customer c, customer_address ca, customer_demographics
WHERE c.c_current_addr_sk = ca.ca_address_sk
AND cd_demo_sk = c.c_current_cdemo_sk
AND Exists (SELECT * FROM store_sales, date_dim WHERE c.c_customer_sk =
ss_customer_sk AND ss_sold_date_sk = d_date_sk AND d_year = 2002 AND d_qoy < 4)
AND (Exists(SELECT * FROM web_sales, date_dim WHERE c.c_customer_sk =
ws_bill_customer_sk AND ws_sold_date_sk = d_date_sk AND d_year = 2002 AND d_qoy
< 4)  OR Exists(SELECT * FROM catalog_sales, date_dim WHERE c.c_customer_sk =
cs_ship_customer_sk AND cs_sold_date_sk = d_date_sk AND d_year = 2002 AND d_qoy
< 4))
GROUP BY ca_state, cd_gender, cd_marital_status, cd_dep_count,
cd_dep_employed_count, cd_dep_college_count
ORDER BY ca_state, cd_gender, cd_marital_status, cd_dep_count,
cd_dep_employed_count, cd_dep_college_count
LIMIT 100
```

Figure 10: The Query35 string.

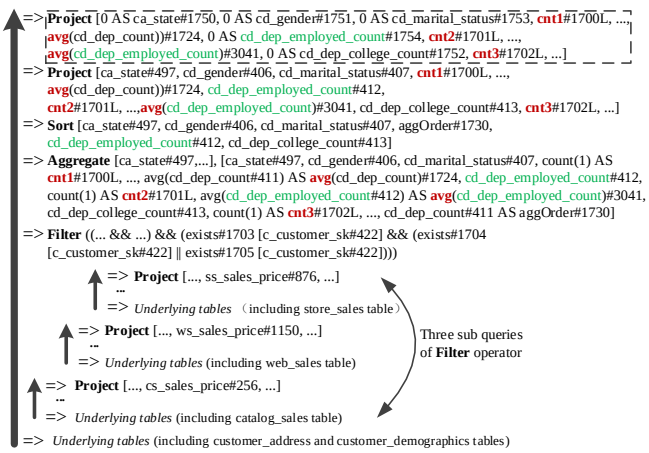


Figure 11: The original analyzed logical plan and secure logical plan of Query35 (Figure 10).

Figure 10 provides Query35 string used in the case study for SQL engine (Section 6.4.1). This query, which contains various data processing purposes, aliases and sub queries, is complicated enough for showing our assessment. Figure 11 shows its original analyzed logical plan and secure logical plan. Figure 12 shows the results about the Query35. The reader will find that some information is omitted in the logical plans in Figure 11. The omitted information cannot obscure the understanding of this assessment. And, it is easy for the reader to find some direct correspondences between the SQL string and its logical plan (e.g., the cd_dep_employed_count corresponds to the `cd_dep_employed_count` in Figure 11.)

As shown in Figure 11, the two plans have a common part including the "Sort" operator and its below operators. The "Project" operator above the "Sort" operator belongs to the original logical plan and another in dotted box belongs to the secure logical plan. The original logical plan makes computation on several objects while outputting these objects (e.g., `cd_dep_employed_count` object, and one of DOP-C purposes on this object is designated by the expression `avg(cd_dep_employed_count#412)` (Section 5.3)).

c_1 : *ca_state*, c_2 : *cd_gender*, c_3 : *cd_marital_status*, c_4: cnt1, c_5: min(cd_dep_count), c_6: max(cd_dep_count), c_7: avg(cd_dep_count), c_8: *cd_dep_employed_count*, c_9 : cnt2, c_{10}: min(cd_dep_employed_count), c_{11}: max(cd_dep_employed_count), c_{12}: avg(cd_dep_employed_count), c_{13}: *cd_dep_college_count*, c_{14}: cnt3, c_{15}: min(cd_dep_college_count), c_{16}: max(cd_dep_college_count), c_{17}: avg(cd_dep_college_count)

Note: The bold italics represent sensitive information. (c_1, c_2, c_3, c_8, c_{13}) The others are computation results.

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}
0	0	0	1698	0	0	0	0	1698	0	0	0	0	1698	0	0	0
0	0	0	1807	1	1	1	0	1807	0	0	0	0	1807	0	0	0
0	0	0	1782	2	2	2	0	1782	0	0	0	0	1782	0	0	0

(a) The results of Query 35 are got from 2GB data with sensitive information directly deleted from these data.

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}
0	0	0	1	0	0	0	0	1	0	0	0	0	1	3	3	3
0	0	0	1	0	0	0	0	1	0	0	0	0	1	5	5	5
0	0	0	1	0	0	0	0	1	3	3	3	0	1	2	2	2

(b) The results of Query 35 are got from 2GB data with sensitive information deleted by zero setting logic in secure logical plan.

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}
null	F	D	1	0	0	0	0	1	0	0	0	3	1	3	3	3
null	F	D	1	0	0	0	0	1	0	0	0	5	1	5	5	5
null	F	D	1	0	0	0	3	1	3	3	3	2	1	2	2	2

(c) The results of Query 35 are got from 2GB data without access control policy.

Figure 12: Three computation results about Query35.

This plan contains many aliases obviously exhibited in the format _ AS _ (e.g., `avg(cd_dep_employed_count)#3041` is the alias of computation expression `avg(cd_dep_employed_count#412)`). Moreover, it includes three sub queries in the "Filter" operator.

The access control policy is that sensitive information in those tables accessed by Query35 cannot be seen. The corresponding objects representing sensitive information in each table can be recognized by initial abbreviation, e.g., `ca_state` object represents state information in `customer_address` table; `cd_gender` represents gender information in `customer_demographics`; `ss_sales_price` represents price information in `store_sales`.

Obviously, in the secure logical plan, all sensitive information is prevented from direct disclosure by the zero setting logic (Section 5.5). The zero setting logic has the format "0 AS _". For example, the "0 AS cd_dep_employed_count#1754" indicates the value of the `cd_dep_employed_count` object is set to zero). Meanwhile, computation results of some sensitive information are disclosed although aliases are used on these results in the secure logical plan. For example, the average of `cd_dep_employed_count` object is disclosed. In addition, those three sub queries try to disclose sensitive information (e.g., price information) to its "Filter" operator; however, the sensitive information is not directly disclosed to a data user and thus we don't use zero setting logic to restrict this disclosing.

The computation results in Figure 12(b) are consistent with ones in Figure 12(c) while outputted sensitive information is set to zero in Figure 12(b). However, the computation results in Figure 12(a) are obviously different from the ones in Figure 12(c) because directly deleting sensitive information from data sources alters computation results.