Membrane – Safe and Performant Data Access Controls in Apache Spark in the Presence of Imperative Code

Andrei Paduroiu
Amazon Web Services
andreipm@amazon.com

Sungheun Wi
Amazon Web Services
sunghew@amazon.com

Yan Yan
Amazon Web Services
yanya@amazon.com

Roni Burd
Amazon Web Services
roniburd@amazon.com

Ruhollah Farchtchi
Amazon Web Services
ruhollah@amazon.com

Giovanni Matteo Fumarola
Amazon Web Services
gifuma@amazon.com

ABSTRACT

Data Governance is an increasingly critical feature of modern cloud database systems, enabling administrators to set granular access policies on their data. AWS customers want to define row or column filtering on their blob storage data and access it using popular tools such as Apache Spark. AWS EMR provides a managed and serverless solution that lets users run Spark jobs in the AWS cloud with imperative and declarative programming against their data, while securely enforcing the fine-grained access controls defined on those datasets. Spark runs its compiler and scheduler alongside the user application and embeds user-defined functions in query plans, giving a threat actor direct access to its memory space. This introduces attack vectors such as information disclosure or privilege escalation during policy enforcement, in addition to well-researched threats such as SQL side channel attacks. In this paper, we present Membrane: a novel approach to secure query plans with declarative and imperative code. The innovation comes from splitting the Spark driver in two in order to rewrite query plans with security boundaries while avoiding traditional tradeoffs when using container isolation techniques. The approach described herein enables applying fine grained data access controls to both SQL and map-reduce Spark jobs, with negligible performance and cost differences.

PVLDB Reference Format:
Andrei Paduroiu, Sungheun Wi, Yan Yan, Roni Burd, Ruhollah Farchtchi, and Giovanni Matteo Fumarola. Membrane – Safe and Performant Data Access Controls in Apache Spark in the Presence of Imperative Code. PVLDB, 17(12): 3813 - 3826, 2024.
doi:10.14778/3685800.3685808

1 INTRODUCTION

Amazon Web Services (AWS) Elastic Map Reduce [4] (EMR) is a public cloud service that provides distributed data processing technologies like Apache Spark [26] and Trino [33], with the latest open table formats such as Apache Iceberg [22]. EMR delivers these technologies to customers using multiple service models including infrastructure as a service (EMR on EC2 [5]), platform as a service (EMR on EKS [2]), and software as a service (EMR Serverless [8]). EMR modifies the Apache Spark open-source engine for AWS customers to bring performance improvements, automatic elasticity and enterprise features such as fine-grained access control (FGAC), that are typically available in mature database systems. Changes made to the engine need to retain, as much as possible, full API and language compatibility with the corresponding open-source version. This paper focuses on the modifications we have made to support FGAC capabilities such as row level filtering and data masking while keeping the full expressivity of the Spark dialect.

Apache Spark uses a data lake pattern of disaggregated storage [57] on top of blob storage systems like AWS S3 [6] or Azure Storage [40] to retrieve and persist customer data in a variety of formats. For governance enforcement, Spark connects to catalog and metadata systems that can resolve objects such as databases, tables and views, and provide permission constructs for security like row/column filters and masking policies. These systems can be open-source, including Apache Hive Metastore [21] and Apache Ranger [24], or open-source compatible like AWS Glue Data Catalog [7] and AWS Lake Formation [9]. The Spark engine retrieves metadata from these systems, materializes schema on read and enforces the applicable security policy as needed. This exchange of information needs to remain secure and immune to tampering or interception by users.

Compared to traditional SQL database systems, Apache Spark allows, as a core feature, mixing of imperative and declarative code and gives users access to a lower-level interface that enable the manipulation of the compiler and resulting query plans, thus presenting design challenges unique to it. To our knowledge, other implementations address this problem by making several trade-offs in their system architecture. One common solution involves using a completely separate filtering fleet (middleware) which adds cost to the customer in terms of extra compute and prevents logical plan optimizations based on the overall query and constraints. A second approach isolates security policies using some form of security boundary with well-known interfaces such as user-defined functions [28] or user-defined aggregator functions [27]; however, this comes with the cost of restricting Apache Spark customer flexibility. Both of these also require choosing a compute isolation model that has to balance security hardening vs query cost in the form of higher row marshaling latency. Several vendors make trade-offs by delineating security along process or container boundaries, or even implement language runtime isolation [39]. For AWS EMR, none of these are acceptable [10, 38] because we believe that only a Virtual Machine can be used as an adequate security container for data policy enforcement protecting from user-provided code,

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685808

which would add significant cost to the customer in form of higher query latency.

The primary innovation presented in this paper describes the process of creating logical isolation boundaries between existing query stages by leveraging natural transitions points such as exchange materializations or shuffles, without requiring the upstream application to adhere to any predefined contracts or behaviors. With some modifications to the compiler, it allows users to write any arbitrary Spark programs with minimal performance penalties for most use cases. The query optimizer takes into account the existing cost of exchanges, the cost of adding security policies and other mitigations for threat vectors common in SQL-based policies while rewriting the query plan. To protect the compiler itself from being tampered with, we present an innovative technique that separates the Spark Driver into a frontend and backend, with the user application running on the former and the full compiler and execution engine on the latter. We also discuss our query plan marshaling process that helps transfer plans between the two and automatically identifies and isolates embedded user code within.

Although the concepts presented here are specific to Spark, we believe these techniques are translatable to many distributed data processing engines with similar architectures as well.

2 BACKGROUND

Apache Spark is used for distributed querying and manipulation of large datasets, enabling data processing with both SQL and programmatic APIs. The entry point for running Spark is the Driver Program, which is a Python, Java, Scala or R program that invokes APIs provided by the Spark library to build and execute a job plan.

The *Spark Driver* [30] is a library referenced by the user-submitted Driver Program that provides the APIs needed to define the job plan. The *Spark Executors* are processes that run the computations (*Tasks*) sent over by the Spark Driver and optionally store intermediate exchange (shuffle) data. Spark plugs into an external *Cluster Manager* that enables it to manage workers for its Executors. Open-source Spark is bundled with integrations for Apache YARN [20, 56], Mesos [23, 34] and Kubernetes [36], while EMR Serverless Spark has a proprietary implementation.

The lowest-level data processing API in Spark is the *Resilient Distributed Dataset* [59] (*RDD*), which implements the map-reduce paradigm [16]. RDDs are composable, and form a directed acyclic graph (DAG). For structured data processing, Spark both supports SQL and provides the *Dataset API* [12], which enables programmatic construction of relational plans. Regular SQL is parsed into a Logical Plan, wrapped into a Dataset, and every Dataset is compiled into an RDD DAG [59] when executed.

The Driver Program triggers the *execution* of a Dataset or of an RDD by invoking an action (such as the *collect()* method) on either. An RDD's DAG is sent to the *DAGScheduler* component in the Spark Driver, which generates a sequence of *Stages* made of parallel *Tasks* [59], further processed by the *Task Scheduler* which orchestrates their execution on the Spark Executors. The Task Scheduler also manages Executors' lifecycle using the configured Cluster Manager. Datasets are executed by analyzing the wrapped *Unresolved Logical Plan*, resolving every symbol to a database object (table, view, function, etc.) and checking for consistency. The resulting *Analyzed Logical Plan* is transformed (using a set of rules) into an *Optimized Logical Plan* and then again into a *Physical Execution Plan*, much like a traditional DBMS pipeline. Spark uses code generation [43] to further transform this plan into an RDD DAG, which is then executed using the procedure described above.

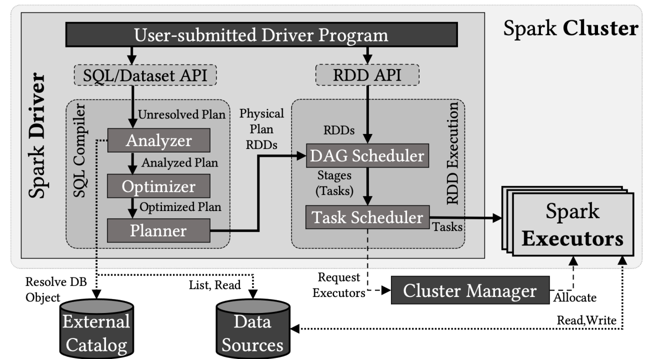


Figure 1: Apache Spark SQL and RDD execution pipelines.

Spark Executors receive Tasks from the Driver's Task Scheduler, containing executable bytecode, that perform a subset of the entire job's computation. Depending on the job, Executors may read data from the *Data Sources* defined in the job plan, may exchange intermediate data between consecutive stages and either write the final stage output data to a persistent store, such as AWS S3 [6], or stream the result back to the Spark Driver for subsequent use by the Driver Program. The entire flow is shown in Figure 1.

3 OVERVIEW

This section covers our design goals, the constraints and challenges we aimed to solve, as well as the high-level architecture of the system. We introduce our general concept and the driving principle behind our design, and deep-dive into specific topics in the subsequent sections.

3.1 Design Goals

We have adopted the following design goals, in order of priority:

Data Security. Enforcing fine-grained data access control policies should not reveal unfiltered data or the policies themselves to users. Dataset owners expect that users with configured policies cannot see records that do not match those policies' filters, whether via a direct projection or as a result of a transformation or aggregation applied to the protected dataset. Furthermore, since the underlying data is stored in systems with coarse-grained authorization controls, such as AWS S3 [6], any credentials granting access to unfiltered data should also not be disclosed to the querying user. This is Membrane's *key tenet*; its design enforces it by using a container or VM to isolate the execution of any user-provided code from the SQL compiler and parts of the execution engine that access raw, unfiltered data or enforce access control policies.

Performance and Cost Parity. Applying security controls typically requires making trade-offs with runtime performance and cost, however our customers expect that doing so should not regress query execution performance by more than 50%. Since both queries

and security filters have very diverse forms, it is difficult to define the general system's performance profile in relation to a baseline. Even with the strict physical isolation described in Section 4 and the logical security boundaries and optimization constraints presented in Section 5, for most queries, Membrane introduces up to a 15% performance impact where no user code is present, and at most 50% when user-provided code is mixed-in in with SQL, as shown in Section 8. It needs a single additional node per Spark cluster to host a secondary Spark Driver, but otherwise the amount of compute required for a job is equivalent to running a similar job without the extra security for data access controls.

Maintainability. Membrane should be compatible with future versions of Spark without the need to refactor with every new release. Its design leverages the existing modularity and plugin architecture present in Spark for injecting custom-designed components that interact with the rest of the system using stable internal contracts, facilitating the effort needed to adapt to new versions.

3.2 Challenges

The previous section discussed Membrane's design goals – what we wanted to achieve. In this section, we summarize the challenges we had to overcome in order to achieve those goals.

3.2.1 Securing the SQL Compiler and Task Planner.

Data security filters for a table are represented as SQL fragments, in the form of predicates or joins, that need to be applied to that table's relation in the logical plan. This can only be done inside the Spark's SQL compiler, when it analyzes an unresolved logical plan for execution. However, the SQL compiler is sharing memory space with the Driver Program, which enables the job submitting user to infer the security policies they are subject to. One potential attack vector is to perform a tree-walk on the rewritten plan, remove the injected filters and submit the resulting plan for execution (bypassing Spark's query execution pipeline), causing the job to run on unfiltered data. The user program may also intercept filtering tasks output from the Task Planner (including DAGScheduler and Task Scheduler) and bypass them as well.

Unfiltered data is protected by a set of credentials which either provide full or coarse-grained access to the dataset. These credentials are different from the submitting user's credentials and can only be retrieved from a trusted store (such as a catalog). Spark needs these credentials to read raw data and perform filtering, and shares them with both the Driver and its Executors, putting them within reach of any user-submitted code running in the cluster.

These can lead to unintended information disclosure [41] or a privilege escalation [48] attack, both of which go against our *Data Security* tenet. Membrane should incorporate a design to effectively secure Spark's SQL Compiler and Task Planner from the reach of the Driver Program.

3.2.2 Protecting from SQL side channel attacks.

A side channel attack [13] is an attempt to exploit side effects of commands with the goal of inferring additional information about the system. In database systems with governance features, query predicates can be crafted to reveal either the presence or the cardinality of data hidden by the filter. Timing attacks [15] are difficult or even impossible to mitigate, while others, such as SQL logical optimization attacks, can be addressed. *Listing 1* presents an example of a row-level filter defined on a table *employee* allowing *user1* to only see rows having *birthday* prior to *2006*. If *user1* wants to infer the existence of rows outside of the filter, they can write a query designed to interfere with the optimization rules present on the system. By forcing a division-by-zero error that would only be thrown if there are any tuples in the table matching the *IF* condition, *user1* can infer the existence of rows in the table that would normally be hidden by the filter.

```
Data Admin:
    CREATE TABLE employees(
           name varchar(50), birthday date, salary int)
    GRANT SELECT ON TABLE employees
          TO user1 FILTER 'birthday<2006-01-01'
user1:
    SELECT * FROM employees
    WHERE 1 / IF(birthday >= '2008-01-01', 0, 1) = 1
```

Listing 1: SQL side channel attack using division-by-zero.

To meet our *Data Security* tenet, Membrane should incorporate techniques to detect and remove logical optimization SQL side channel attacks when enforcing fine-grained data access controls.

3.2.3 Sandboxing user code execution.

User-defined functions [12, 28] (UDFs), user-defined aggregator functions [27] (UDAFs) and user-defined types [12, 32] (UDTs) are popular features supported by Spark's Dataset API [12], extending the expressivity of built-in Spark functionality with user-specific business logic. Spark users can provide programmatic (non-SQL) implementations for any of these, and then use them in Dataset transformations. When referenced, Spark embeds the binary code of a UDF or UDAF as specialized logical plan nodes, and creates transformation nodes for UDTs to/from Spark's internal representation (using the UDT's provided serialization code). The binary code is further serialized (if Java/Scala) or pickled [49] (if Python).

When compiled into a Task, there is no differentiator between such user-provided code and code that was generated by Spark's SQL Compiler. Such user code has access to the entire process space of the Spark Executors that run the enclosing Tasks, which puts any credentials present on those executors within their reach. The code within the UDF/UDAF/UDT can take full control of the Executor and use those credentials to access unfiltered data, leading to a privilege escalation attack. With respect to RDDs, almost every implementation wraps user-provided code so they are susceptible to the same problems described above.

Membrane should detect user-provided code embedded in query plans in the form of UDFs, UDAFs or UDTs, or wrapped in RDDs, and prevent them from making unwanted changes to query plans or access credentials or data, in line with our *Data Security* tenet.

3.3 Architecture

In this section we describe Membrane's general system design. The approach presented here directly helps solve two of the challenges listed in the previous section, namely *Securing the SQL Compiler and Task Planner* and *Sandboxing user code execution.*

Membrane splits the Spark Cluster into two partitions. The *User Space* runs Spark components (one driver and zero or more executors) with the submitting user's credentials. The *System Space* has

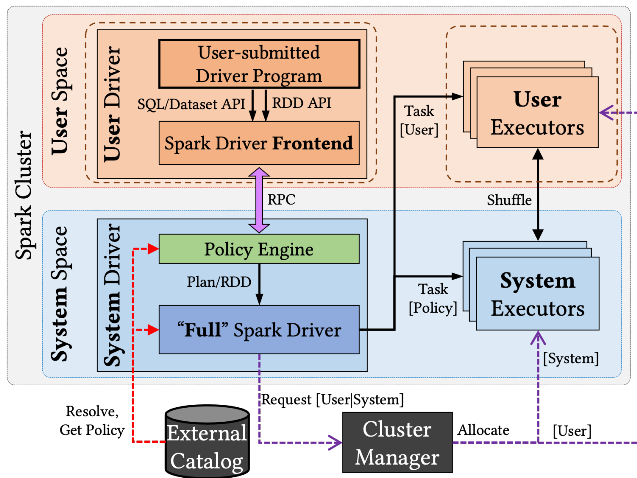


Figure 2: Partitioning the Spark Cluster into User Space and System Space.

Spark components (one driver and zero or more executors) running with special *service credentials*. These service credentials carry over the submitting user's identity, but cannot be created by the user themselves (they are "system"-generated).

This split aims to achieve sandboxing at the stage boundary (instead of user code boundary): we want to have all types of user code (Driver Program, UDFs, UDAFs, UDTs and user-submitted RDDs) execute only on components in the User Space and reserve the System Space for operations that deal with enforcing data access controls and filtering. It is important to note that even if there are now two Spark Drivers, we do not use the System Space merely as a "filtering fleet" that sits between the raw, unfiltered data and the user Spark Cluster. Both the User and the System Spaces are part of the same Spark Cluster and all the components within contribute towards the execution of the Spark jobs submitted to it, with only one driver managing it. This is key to achieving our *Performance and Cost Parity* goal as outlined in Section 3.1.

We can observe the lifecycle of a Spark job in Figure 2. The *User Driver*, sitting in the User Space, is where a Spark job begins. The Driver Program runs here, instantiating a Spark library and invoking Dataset or RDD APIs, exactly as it would do in a normal Spark Cluster. However, this Spark driver is only a *frontend*; we modified it to intercept the SQL Compiler pipeline and the RDD Execution pipeline and extract Logical Plans and RDD DAGs, which are then sent through RPC calls to the *System Driver*.

The *System Driver*, part of the System Space, has an RPC service listening for requests from the User Driver. Upon receiving one, it rebuilds either the original logical plan or RDD DAG, runs it through a *Policy Engine* (which cleanses and rewrites it to enforce data access policies), and then passes the resulting plan/DAG to the *real* Spark Driver instance. Only the System Driver is connected to the External Catalog, Data Sources and Cluster Manager, therefore only it may resolve catalog objects, request executors from the Cluster Manager, or otherwise communicate with executors. The User Driver is not allowed to do any of these actions.

Executors are of two types. *System Executors* run in the System Space and they may only execute tasks that read data, enforce filters or otherwise do not invoke any user code. *User Executors*, conversely located in the User Space, run all tasks that may invoke user code, but cannot run any policy-related tasks.

We modified the *Planner* inside the System Spark Driver to label tasks based on what they are doing; the Planner uses lineage techniques to determine the provenance of the code inside each task – information only available within the Spark Driver. Tasks may thus be labeled either as *User* or *System*. We mapped these to Spark's *Resource Profiles* [18] by creating a System and a User profile, respectively, and then leveraged the existing mechanism inside Spark to run them in the appropriate executor.

We modified AWS EMR's Control Plane to start Spark clusters with two drivers, enforce firewalling around User Space components, provision containers with the correct credentials, and only allow the System Driver to request resources.

The following sections detail each design vertical that compose Membrane's architecture. We begin with a deep-dive into how the Spark driver was split, followed by query plan manipulation techniques leading to affinitized query execution, and close with an experimental evaluation of this design on AWS EMR.

4 DRIVER SEPARATION

Membrane sandboxes the Driver Program by using two drivers. The *User Driver* acts as a *frontend* to the Spark cluster. It executes the Driver Program, builds one or more Logical Plans or RDD DAGs as outcomes and delegates their execution to the *System Driver*, which is the actual coordinator of the entire Spark Cluster. The execution result is sent back to the User Driver and provided to the calling code - fully unaware of the underlying mechanism. Figure 3 visualizes the separation process and highlights the added components to bind the two drivers together.

4.1 User Driver

The User Driver is similar to the Driver in a non-Membrane-enabled Spark Cluster. It runs inside a container (Apache YARN [20]/AWS EKS [2]/AWS Fargate [3]/etc.) setup by the AWS EMR Control Plane, which also configures several firewall measures around it. It is launched by either invoking the *main()* method (Java/Scala) or running a Python script, after which it is free to call any Spark APIs available to it. The AWS EMR Control Plane pre-configures Spark in this container with delegated SQL and RDD execution pipelines, as well as a Catalog Proxy in lieu of the actual catalog client.

For Dataset execution, we made the conscious decision to intercept the pipeline immediately after the analysis phase. Although intuition suggests that we should have sent unresolved plans to the System Driver and do the entire analysis there, we found this would lead to an extremely chatty protocol due to Spark performing plan analysis upon every transformation of a Dataset. Prior to execution, a Dataset is transformed multiple times until finally executed, so by taking analyzed plans we reduced the over-the-wire calls from an unbounded number down to a single one per execution, all without affecting the integrity of the query. We override Spark's *QueryExecution* class [31] to fetch the Analyzed Plan, marshal it

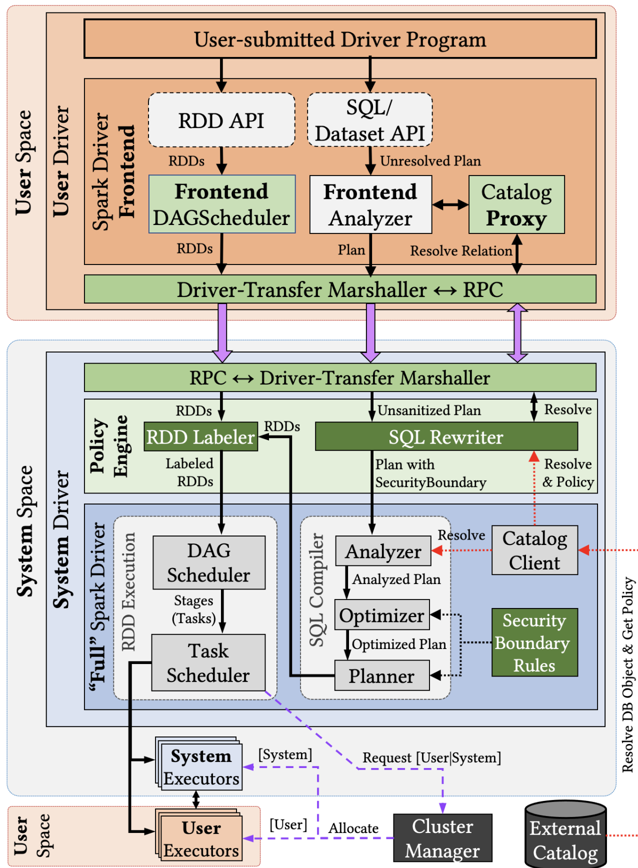


Figure 3: Driver Separation. Logical Plans and RDDs are intercepted on the User Driver and sent to the System Driver, where they are run through a Policy Engine and handed off to the underlying Spark execution pipelines.

into a wire-friendly format and delegate its execution to the System Driver via an RPC call.

This introduced another challenge: since analysis is partially performed on the User Driver, Membrane needs a way to resolve database object names without direct access to the External Catalog. We solved this by creating a *Catalog Proxy* that delegates object name resolution via RPC to the System Driver.

The RDD execution is simpler. As shown in Figure 3, Membrane only had to modify the *DAGScheduler* to marshal the user-submitted RDD DAG into a wire-friendly format and delegate its execution to the System Driver via RPC.

4.2 System Driver

The System Driver is a *service* running in a container setup by the AWS EMR Control Plane, with the RPC endpoint listening for requests from the User Driver being its only entry point. No Driver Programs can be submitted or executed here and no other connections other than the one from the User Driver are accepted.

As shown in Figure 3, there are two independent layers inside the System Driver. The *Policy Engine* sits immediately below the RPC endpoint, with all Logical Plans and RDDs coming through being first reconstructed using Membrane's *Driver-Transfer Marshaller* (Section 4.3), and then processed to sanitize them and enforce any applicable data security policies.

The *RDD Labeler* handles RDD DAGs coming through the RPC endpoint and labels those RDDs with the *User Resource Profile*. Since their contents are opaque, Membrane cannot reason whether the bytecode wrapped inside RDDs is safe or was designed to carry information disclosure or other types of attacks. As such, no user-submitted RDDs may be assigned the *System Resource Profile*, however some internally-generated RDDs for which we have clear lineage (from the *SQL Compiler*) may be labeled as such.

Incoming *Logical Plans* and *Resolve Relation* requests are run through the *SQL Rewriter*, which applies security policies to them and prevents such policies from leaking out when the User Driver requests an *explain plan* or wants to resolve an object name. Policies are applied by substituting a logical relation pointing to a protected resource with a *Security Boundary* node (Section 5.1). When a logical plan needs to be sent to the User Driver, whether to resolve a relation, or as part of the *explain* command, the SQL Rewriter replaces the Security Boundary node with a *Remote Logical Relation* node containing only the name of the resource and the columns visible to the user, thus preventing disclosure of the actual security predicate or the full underlying schema. If this plan is received again (as part of a Dataset execution), the Remote Logical Relation is substituted back with the original Security Boundary node.

The Policy Engine outputs *Labeled RDDs* and *Logical Plans with Security Boundaries*, sending them to either the RDD Execution or SQL Compiler pipelines. The System Driver runs an unaltered instance of Spark with no modifications to its components. Using existing contracts, Membrane latches onto the *SQL Optimizer* and *Planner* to introduce additional rules and constraints to prevent SQL side channel attacks. The Planner also labels its output RDDs with either the *User* or *System* Resource Profile, depending on what role they play in the query execution (using the *RDD Labeler*).

The rest of the pipeline runs plans through the classical Analyzer-Optimizer-Planner sequence, and both the output RDDs and the labeled RDDs received from the User Driver are sent to the DAGScheduler and finally the Task Scheduler. No more changes were needed here as Spark already supports resource profiles; it makes requests for the appropriate type of executors from the *Cluster Manager* and routes Task execution to executors with matching resource profiles.

4.3 Driver-Transfer Marshalling

The driver split sandboxes the Driver Program from the Spark SQL Compiler and RDD Execution pipeline, but it introduces the challenge of *transferring* Logical Plans and RDD DAGs from the User Driver to the System Driver. Even if marked as serializable, the classes we need to transfer were not designed to be marshaled over to another Spark driver (the intent was to copy them from the driver to executors). Native object serialization [46] cannot be used here because it restores the entire inner state of an object and thus may pick up unwanted objects like dynamically-generated lambda methods or object state specific to the JVM of the User Driver.

We designed the *Driver-Transfer Marshaller* to take an arbitrary Spark Logical Plan or RDD DAG, convert it into a wire-friendly

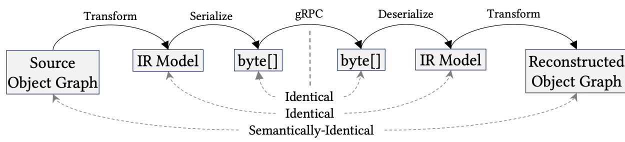


Figure 4: Driver-Transfer Marshalling transforms an object into a model, which is serialized to/from a byte array, and then transformed back into a semantically-identical object.

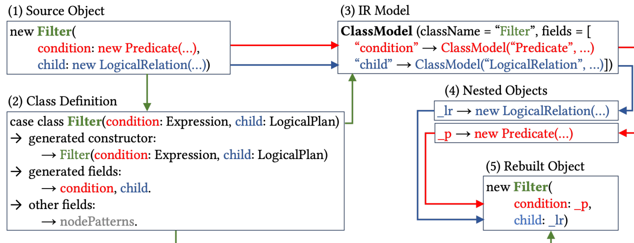


Figure 5: Transformation example for the Filter class (1). The constructor and relevant fields are identified (2), a model is created from the class definition and source object (3), with fields recursively transformed. Unmarshalling rebuilds nested objects (4) and invokes the constructor with them as arguments to instantiate the rebuilt object (5).

format, and reconstruct a semantically equivalent [52] plan or RDD DAG, with no structural changes, that produces the same result.

The marshalling process (Figure 4) transforms an object graph into an *intermediate representation* (model) that describes it, and then serializes that model into a *byte stream* that can be transferred over the wire. The model is made up of a handful of primitive structures indicating the class of the object, primary fields, etc., with specialized model types for collections, maps or other corner cases such as singletons.

The *object-to-model transformation* relies on Scala case classes [51] invariants: immutable classes, with the primary constructor's arguments becoming class fields with the same names. Since most of Spark's classes fit this description, we have designed the transformation process to identify the primary constructor of a class, extract its argument names and then look up only the fields with the same name and type. In the model for that object, we record the class name and the transformed model for objects pointed to by each of the chosen fields. When we unmarshall, we first transform the inner fields, and invoke the class's constructor with the inner field values as arguments (matching on name and type). Figure 5 uses Spark's *Filter* class [29] as an example to visualize this process. This method works for most of the classes in Spark's logical plans and RDDs. We created specialized transformations for collections, maps, enums and other one-offs, and manual *data transport classes* for those handful of classes that couldn't be handled automatically.

Embedded user-defined code, such as UDFs, UDAFs, UDTs and RDD functions may require lambda serialization [45] and cannot be marshalled using the method above. We use the native Java serializer [46] for them, however extra care must be taken as the

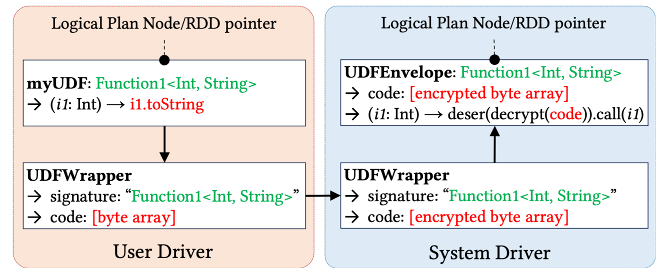


Figure 6: UDF handling example. A function has its signature extracted and is closure-serialized on the User Driver. On the System Driver, the serialization is encrypted, a strongly-typed envelope is created based on the signature and substituted for the original UDF in the Logical Plan.

simple deserialization of this code can introduce attack vectors such as *serialization gadgets* [53].

As exemplified in Figure 6, embedded user-defined code is serialized on the User Driver into a byte array and bundled with some metadata describing its *signature*. Both these are sent to the System Driver, which encrypts the byte array and wraps it into an envelope that matches the previously extracted signature. The signature describes the user code wrapped: UDFs and RDD functions contain the number and types of arguments along with the return type, UDAFs contains the Input/Output/Buffer data types, and UDTs contain the inner type name. The SQL compiler uses this metadata to validate the plan integrity and generate correct code. The envelope requires the *decryption* key when invoked; since this key is only present on User Executors, it prevents unintended deserialization and execution of user-provided code on System Space components.

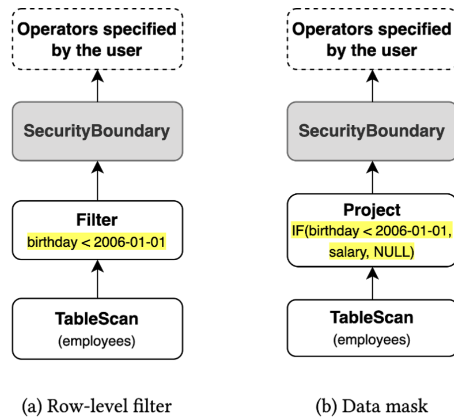
5 QUERY PLAN MANIPULATION AND OPTIMIZATION

This section describes how data security filters are injected into a query plan using a new *Security Boundary operator* that helps prevent SQL side channel attacks (Section 3.2.2), as well as some performance-oriented optimizations applicable to this operator that do not weaken its security constraints.

5.1 Injection of Data Security Filters

The System Driver's SQL Rewriter injects a data security filter into a query plan when a plan symbol resolves to a protected table (Section 4.2). Doing so in the System Space ensures that the injection logic cannot be overridden or bypassed by threat actors. A Security Boundary operator, newly introduced in Membrane, is added with data security filters to enforce security constraints.

Figure 7 (a) and (b) illustrate a sample query plan with the highlighted security filter after a row-level filter and data mask are injected, respectively. The row-level filter is represented as a normal *Filter* operator, while the data mask is represented as a *Project* operator with an *IF* expression. In the data mask example of Figure 7 (b), only the *salary* information of accessible *employee* rows is visible while the *salary* information of other rows is masked as *NULL*. Row-level filters and data masking may coexist for the same



(a) Row-level filter (b) Data mask

Figure 7: Example query plans with the injection of data security filters.

table, in which case both a *Filter* and a *Project* operator are created together for the same table. All data security filters are evaluated under a security boundary, ensuring that data the current user is not privy to does not go beyond that operator. Any other operators specified by the user are located above the security boundary, and they are executed only for filtered data.

Column-level filters are enforced by means of a *Project* operator. When resolved to a protected table, the underlying table relation is wrapped in a *Project* operator containing only the authorized columns; thus, any upstream operators are limited to referencing the set of columns defined within.

5.2 Security Constraints Guarantee

The principal reason for introducing a security boundary operator in a query plan is preventing unwanted plan optimizations from being applied across the boundary itself. By clearly delineating the data security filters from the user-specified operators, it removes opportunities for unintended information disclosure.

Let's recap the SQL side channel attack example from *Listing 1*. To ensure that *user1* cannot infer if an *employee* whose *birthday* is in an inaccessible range exists, the row-level security filter has to be evaluated prior to the *IF* filter specified by the user. Therefore, the divide-by-zero error will not occur because the *IF* user filter is executed only for already-filtered data. In the *Spark Query Optimizer* [12], each optimization rule is applied to a specific pattern of a sub-plan by explicitly specifying the target pattern such as operator type and its property. The security boundary operator, located between data security filters and user-specified operators, prevents existing optimizer rules from being applied across the boundary because no existing optimizer rule specifies a security boundary operator as a target pattern. This guarantees that no operator specified by a user may exist under the security boundary and the security filter is evaluated prior to any user operators. However, all existing optimizer rules can still be applied inside each boundary because doing so does not risk breaking Membrane's *Data Security* tenet. For instance, a user filter that exists at the top can be pushed down until a security boundary node is met, and the same is applied to inside that security boundary. A security filter that exists inside a security boundary can be further pushed down to a scan operator to efficiently prune unnecessary partitions.

Security boundary operators also help with not exposing the definition of the security filter itself. For example, the *explain* feature is designed to show the entire query plan and may reveal the definition of the security filter. With Membrane, the security filter is obfuscated to not make it visible to an unauthorized user in the *explain* result. Security boundary operators make tracking data security filters trivial; this would be intractable otherwise because of various transformations applied to the plan as a result of existing and future optimizer rules present in Apache Spark.

```
SELECT * FROM employees
INNER JOIN filter_table
ON employees.birthday = filter_table.birthday
```

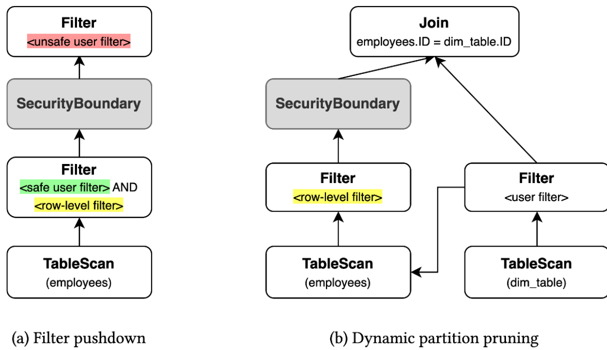
Listing 2: SQL example where the filter derivation reveals the definition of a security filter.

The security filter definition may also be exposed by newly derived filters from a security filter. Considering the example in *Listing 2*, using the same *employees* table from *Listing 1*, a threat actor crafts a join query between the protected *employees* table and an arbitrary table to have the join condition and the row-level filter on the same column (*birthday*) of the *employees* table. Mixing the row-level filter *(employees.birthday < 2006-01-01)* with the join condition (*employees.birthday = filter_table.birthday*), the additional filter (*filter_table.birthday < 2006-01-01*) can be now derived for the joined table to prematurely filter out records that do not match the join condition. This is a well-known optimization technique [47] available in Spark which causes the newly derived filter to be shown in the query plan and therefore exposed to an unauthorized user. The user can infer which security filter is defined by trying various queries, changing the join condition for each column and investigating the resulting query plan because the new filter can be derived only if the join condition and the row-level filter are defined on the same column.

The filter derivation rule in Spark combines join conditions and effective filters from each child of a join operator to discover new derivations. In the *Listing 2* example, the additional filter can be derived for *filter_table* by combining the join condition and the row-level filter returned from a child of a join operator. In Membrane, the security boundary operator does not pass the information of security filters to its parent operator, preventing them from participating in filter derivation. Security Boundary operators also make it easier to disable the derivation.

5.3 Application of *Safe* Optimizations

The previous section describes how Membrane guards against SQL side channel attacks by introducing a security boundary to disable cross-boundary optimizations. However, executing it as-is will drastically degrade the query runtime performance because no optimizations are allowed across the boundary. This section describes how Membrane achieves its *Performance* goal while maintaining the *Data Security* tenet by identifying and applying safe optimization rules to the security boundary. The applicable rules may vary depending on queries/applications, and three important and commonly used rules are described.



(a) Filter pushdown

(b) Dynamic partition pruning

Figure 8: Query plans where safe optimizations are applied.

5.3.1 Filter pushdown.

Filter pushdown [14] is one of the most important optimizations in query engines. Membrane identifies safe user filters that do not risk disclosing information on unauthorized data even when pushed down through a security boundary. A user filter is considered safe if none of the following conditions hold for all expressions included in the filter:

- May trigger exceptions for specific input values or ranges (which could be inferred if an exception occurs).
- Enable tracing of input values or related information into a user-accessible place (i.e., cloud storage, log file).
- Contain user-defined code such as UDF - the implementation of user-defined code is opaque to Spark.

For example, the simple filter *column = <constant literal>* is considered safe while the user filter in *Listing 1*, which is *1 / IF(birthday >= '2008-01-01', 0, 1) = 1*, is considered unsafe because a divide-by-zero exception can occur for a specific value range. Membrane maintains a non-overridable, hard-coded list of safe filter types. Once safe user filters are identified, they are pushed down through a security boundary and can be freely mixed up with injected security filters. If a user provided filter is a conjunctive filter that is a mixture of a safe and an unsafe filter, only the safe part of the user filter is selectively pushed down while the unsafe filter still remains over the security boundary, as illustrated in Figure 8 (a).

5.3.2 Dynamic partition pruning.

Dynamic partition pruning [11] (DPP) is a necessary optimization for a join query in a massively parallel processing system like Spark because it can efficiently filter out large portions of tables and thus significantly reduce the data transferred over the network. For example, when a large fact table is joined with a smaller dimension table that has a selective filter, 1) the selective filter on the dimension table is evaluated first; 2) the small list of filtered values for a join column is transferred to the fact table; and 3) the transferred values are used to efficiently prune unnecessary partitions of the fact table. This is a common pattern of query plans for a star schema [17].

Figure 8 (b) illustrates an example query plan where the protected *employees* table is joined with *dim_table*, and the safe DPP optimization is applied across the security boundary for the *employees* table. The DPP filter is semantically equivalent to the filter of *<join-key> IN (<values>)* where the values are determined at runtime from another table; in this example it is *employees.ID IN (<values from dim_table.ID>)*. The safeness of the DPP filter is determined by checking the semantically equivalent *IN* filter with the definition in Section 5.3.1, and ensuring that all three included expressions (column access, *IN*, and constant literal) are safe.

5.3.3 Projection pushdown.

The projection pushdown [12] optimization technique prunes unnecessary columns as early as possible by pushing down a project operator. To determine if it is safe to push down a project operator through a security boundary, all expressions in the project operator are checked following the same definition described in Section 5.3.1. Unsafe expressions are evaluated outside a security boundary since they cannot be pushed down. Instead, the referenced columns are extracted from the unsafe expression, and an additional project operator with those columns is pushed down through the security boundary so that any unnecessary columns can be pruned. For example, *1 / IF(birthday >= "2000-01-01", new_code, old_code)* is an unsafe expression that needs to be evaluated outside a security boundary, and a new project operator with the referenced columns *(birthday, new_code, and old_code)* is pushed down through the security boundary.

6 PHYSICAL EXECUTION

This section explains the security constraints imposed when a task is assigned to an executor and how Membrane sandboxes user code execution (Section 3.2.3) by appropriately labeling executors and performing necessary stage separations.

6.1 Executor Security Constraints

Spark splits input data into *partitions* and distributes them to multiple executors so that they can be concurrently processed for massive parallelism. When the input data is partitioned, it is usually aligned with group-by columns or join conditions to avoid an additional merge step. If the intermediate results need to be re-partitioned for a subsequent join or aggregation, they are stored on disk and transferred to another executor after being repartitioned with another group-by column or join condition. The operator responsible for this partitioning is called *shuffle exchange*, and the corresponding query plan fragment, executed locally in an executor prior to it, is called a *stage*. Each stage is made up of multiple *tasks* responsible for processing each partitioned data, with the same query plan fragment being performed in all tasks within the same stage. Tasks are then assigned to proper executors by Spark's *Task Scheduler*.

Membrane imposes executor *security constraints* when a task is assigned to an executor. Tasks containing user-defined code, such as UDF, UDAF, UDT or user-submitted RDD, have to be performed in a *user executor* while the unfiltered data for which a security filter has not been applied yet can only be accessed in a *system executor*. This is needed because user-submitted code may gain access to the unfiltered data if it were executed in the same executor (Section 3.2.3). The goal of the executor security constraints is to enforce a container-level separation of unfiltered data processing from user-submitted code execution that may be under control by external threat actors. However, when conflicting operations coexist in the same task, these constraints cannot be met as-is because a task is the smallest unit of work that can be scheduled to an executor. The

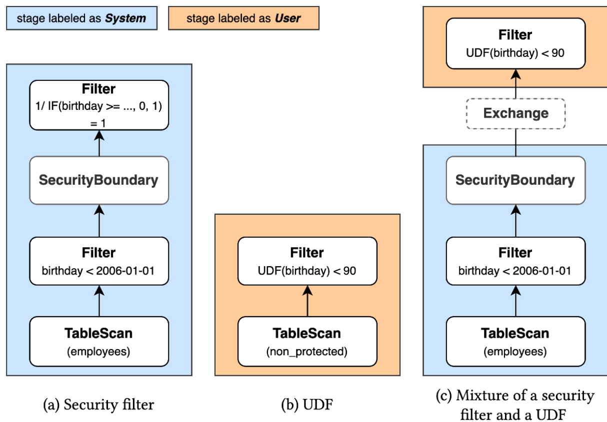


Figure 9: Examples of executor type labeling.

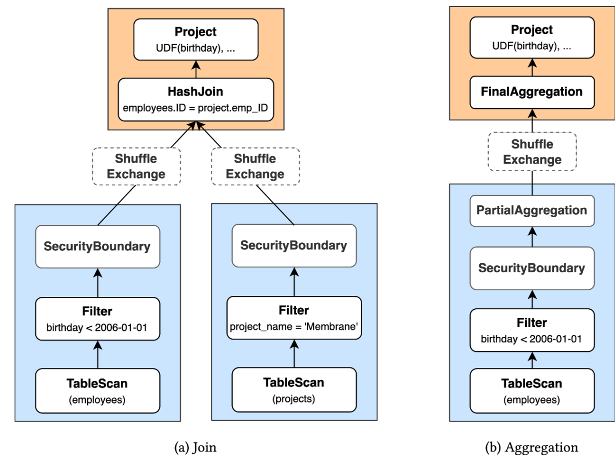


Figure 10: Examples of avoiding unnecessary stage separations for executor type labeling.

task that contains the conflicting operations needs to be split into two. If the data security filter has already been applied to the input data and if there is no user code in a task, it does not violate the constraints to perform it in either executor type. It can be freely adjusted for better performance, and the optimization made by exploiting this property is further explained in the next section.

All tasks within a stage perform the same part of a query plan so they have the same executor security constraints. As such, the decision on the proper executor type is made at the stage level, not task level. To meet the executor security constraints, Membrane performs the following work:

- The proper executor type (either *System* or *User*) is labeled for each stage.
- The physical query plan is adjusted with necessary stage separations to avoid having conflicting operations in the same stage.

6.2 Executor Type Labels and Stage Separation

Figure 9 illustrates how each stage is labeled for simple query plans containing a security filter, a UDF, and its mixture, respectively. In Spark, these query plans can be executed in a single stage, and it is the same in Membrane for the examples of Figure 9 (a) and (b). Each rectangle in the figure denotes a stage boundary, with the stage containing a security boundary being labeled as *System* because it includes the unfiltered data access. An additional observation is that even the filter located outside the security boundary, previously treated as unsafe from a side-channel perspective as explained in Section 5, can be executed in a system executor because it consists of only built-in functions whose implementations are hard-coded inside Spark/Membrane and out of control from threat actors.

There can also be cases where multiple stages exist under a security boundary if the security predicate is not a simple filter. For example, if the security filter is a sub-query that refers to another table (e.g., *employee_id IN (SELECT employee_id from <allowlist_table>))*, it can be internally translated into a join by the Spark Optimizer and result in multiple stages. In this case, all stages under the security boundary are also labeled as *System* because the evaluation of the security filter is not completed yet.

The stage containing a UDF filter is labeled as *User* as illustrated in Figure 9 (b). Since the table used in this example does not have any defined data security filter, the UDF filter can be executed in the same stage as the table scan, and the stage is labeled as *User*.

In contrast, the query plan of Figure 9 (c) has both a data security filter and a UDF filter, which have conflicting security constraints. Since they cannot be executed in the same stage, the stage is split in two, with the stage containing unfiltered data processing labeled as *System* while the stage with a UDF filter labeled as *User*. These two separate stages are connected via an exchange operator that transfers intermediate results from one to the other. This additional stage separation and the corresponding intermediate result transfer caused by the executor security constraints are minimized in Membrane and occur only when necessary. The exchange operator added to meet the executor security constraints differs from a normal shuffle exchange operator in that it doesn't need to repartition data and can arbitrarily distribute the data in an efficient way instead, which would be beneficial for a skewed data set.

Figure 10 illustrates how Membrane avoids unnecessary stage separation caused by the executor security constraints for two of the most commonly used operators, a join and an aggregation when a projection with a UDF is located at the top. In the join example of Figure 10 (a), two tables protected with a security filter each are joined by Spark's shuffle hash join algorithm that partitions both tables by a join column and performs a join within each partition where the records with the same join key are co-located. There are three stages with two shuffle exchange operators located at the stage boundaries (same as classic Spark). Since the projection with a UDF exists in a different stage from the unfiltered data access, the top-most stage is labeled as *User* while the other two table scan stages are labeled as *System*. This enables meeting the executor security constraints even without additional stage separation, and the query plan is the same as classic Spark.

Let's look at another aggregation example in Figure 10 (b) where we have a projection with a UDF at the top. In Spark, the physical execution of an aggregation has two parts – partial aggregation and

final aggregation. Executing a partial aggregation in the same stage as the table scan reduces the transferred data size by executing the aggregation earlier. Since the data is not partitioned by group-by columns in the scan stage, the records for the same group may be spread over different tasks, requiring an additional merge step in the final aggregation operator, after the partially aggregated result is shuffled by grouping columns in the shuffle exchange operator. Similar to the join example, the UDF projection is executed in a different stage, therefore it meets the security constraints to label the stage *User* and there is no need to separate the stage.

Real-world Spark analytic queries are likely to consist of multiple stages due to either a join and/or an aggregation, so the forced stage separation described above is not a common scenario. As shown in Figure 10, additional stage separation is not needed unless any user-defined code is located in the same stage as the unfiltered data access that usually happens only in a leaf stage.

Another optimization made during executor type labeling is minimizing the transition from one executor type to another, since this process introduces additional overhead for deallocating old and provisioning new executors. In cases where a query contains either a security predicate or user-defined code (but not both), all stages are labeled as either *System* or *User*. This is done by exploiting the property that the stage with neither a security predicate nor a user-defined code can be labeled as *anyone* as explained in the previous section. Depending on the other stages in the query plan, the labels for those stages can be adjusted to avoid unnecessary executor type transition. If a query is a mixture of a security predicate and a user-defined code, the transition from one executor type to another type is inevitable, and it is optimized to minimize the number of transitions. For example, the scan stage with a security filter is labeled as *System*, and the following stages can also be labeled as *System* until a stage with a UDF is met. Once the stage is switched to *User* for a UDF, the following stages can also be labeled as *User* to avoid additional transitions unless there is conflicting operation.

6.3 Task Assignment to Executors

Once each stage is labeled with a proper executor type, assigning tasks created from a stage to proper executors relies on Spark's *Resource Profile* feature [18]. Originally introduced for stage-level scheduling, resource profiles enable a user to choose different types of executors for each stage depending on its resource needs (e.g., CPU and memory). This mechanism is reused to meet Membrane's executor security constraints. Membrane introduces a dedicated resource profile for *User* and *System* executors each, and the proper resource profile is attached to a task referring to the executor type label available in each stage. The Spark Task Scheduler then assigns it to the appropriate executor.

7 ADDITIONAL SECURITY MEASURES

The design described in this paper is only part of the fine-grained access control enforcement in Spark running in AWS EMR. However, by itself it cannot cover all possible attack vectors in the environment. Some additional measures that support the safety of a privileged environment handling access-controlled data and credentials, enabling it to coexist and communicate with a user-tamperable environment, are summarized below.

Physical separation of containers. Using VMs or physical EC2 [5] nodes instead of containers is a safe method to prevent a *User* space *container breakout* [38] from affecting *System* space components, but it comes at an increased cost to the customer. To keep the flexibility offered by containers and also isolate *User* space components, the EMR Control Plane creates a physical boundary between the two spaces, using a variety of methods. One example of such a method may involve enforcing that no *User* space container can share a VM/EC2 host with any *System* space container or vice-versa.

Encryption, authentication and authorization. To prevent a threat actor from sniffing data on or impersonating the system driver or system executors, all at-rest and in-transit data are encrypted with separate keys, and server endpoints communications involving such nodes require authentication and authorization.

Access control enforcement during marshaling. Various access control mechanism including file system and network access control during un-marshaling, combined with an allow-list of safe classes that can be deserialized on system nodes are enforced to prevent unintended remote code execution.

8 EXPERIMENTAL EVALUATION

In this section, we outline three experiments that showcase the performance characteristics of Membrane, and compare the run time results with "baseline" Spark (AWS EMR Spark without Membrane changes). The experiments are designed to measure the performance implications from:

- Having two Spark drivers, in Section 8.1,
- Presence of data security filters, in Section 8.2,
- Presence of data security filters and UDFs, in Section 8.3.

In all experiments, we ran the TPC-DS [42, 55] performance benchmark at 3 TB with partitioned Hive tables, against Apache Spark 3.4.1 code deployed on AWS EMR Serverless 6.15. Unless specified otherwise, all clusters were launched with a fixed number of 20 executors, each with 4 cores and 10 GB memory. For the Membrane-enabled clusters, all 20 executors were system executors. As Membrane requires both at-rest and in-transit encryption, for an equitable comparison we also enabled them for the baseline runs.

8.1 Overhead from Membrane Design

We ran the full TPC-DS benchmark against tables with no security filters configured and compared the physical plan shapes between the same queries on both runs, confirming that the query planning and execution process was identical on both Membrane-enabled and baseline Spark clusters. Membrane showed a 5.2% increase in total execution time, broken down in two categories. Up to 2 seconds per query was due to the User-System driver separation (Section 4), positively correlated with the query complexity, number of relations involved and, to a lesser extent, the size of the query result. The rest was attributable to the access control enforcement discussed in Section 7 and was influenced by the total size of the data exchanged throughout the query execution.

8.2 Impact of Data Security Filters

This scenario represents the case of querying a table that has data security filters enabled, in which case Membrane needs to introduce

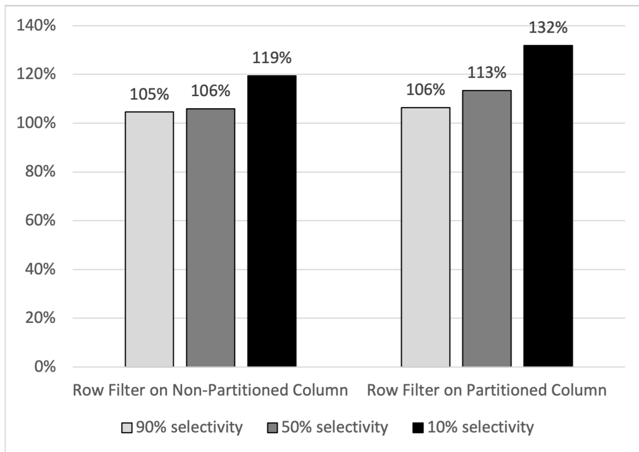


Figure 11: Total execution time percentage comparison of Membrane to baseline Spark for TPC-DS queries using the *store_sales* table, with a data security filter on non-partitioned and partitioned columns and various selectivity levels.

additional security boundaries as described in Section 5 to perform data filtering before transferring the intermediate data to the user accessible environment.

If the filter prunes out a sizeable portion of the table, the resulting smaller dataset may require less time to process and hence compensates for the extra time introduced by the security boundary constraints. To remove this variable from the experiment, we modified the baseline TPC-DS queries by inserting a predicate equivalent to the data security filter, which ensures that the output data is the same and only the overhead from the security boundary is captured.

We have selected the biggest fact table in the TPC-DS dataset, *store_sales*, to apply the data security filter. To emulate various customer workloads, we evaluated the performance for three different security filter selectivities: 10%, 50%, and 90%, which indicates how much data from the table remains after filtering. We repeat the experiments with two different security filters: a row filter on a non-partition column, *ss_item_sk*, in the form of *ss_item_sk <= <value>*, and one on a partition column, *ss_sold_date_sk*, in the form of *ss_sold_date_sk <= <value>*. We then run all 70 queries that refer to the table *store_sales* and compare their execution time with that of the baseline run. We have also tried 0% and 100% selectivity level filters, but we did not include them in this evaluation since the results were difficult to compare with the baseline. The former cuts short the query execution after applying it, while the latter was close to not having a filter at all.

Security boundaries can benefit from the same safe optimizations present in the baseline, with examples shared in Section 5.3. Figure 11 shows that for the 90% selectivity filter, Membrane adds a 5% overhead for the non-partitioned column case and 6% overhead for the partitioned column case. The 10% selectivity runs have the most performance impact among all benchmarked levels, due to the filter derivation from a security filter being intentionally disabled as explained in Section 5.2, with the impact of the missing filter becoming higher as the selectivity decreases. Assuming that the row filter is *ss_item_sk <= 324000* and the query has a join condition *store_sales INNER JOIN items on ss_item_sk = i_item_sk*, in baseline Spark the filter *i_item_sk <= 324000* will be derived on the item table because the join key is matched to the filter condition. While for the 90% selectivity case only 10% of the data is additionally processed on Membrane as the result of disabling this optimization, this effect is much more prominent for the 10% selectivity case on baseline Spark, widening the performance gap. This is a result of deliberately prioritizing our *Data Security* tenet above goals such as *Performance Parity*, when having to make design trade-off choices.

8.3 UDF Placement Implications

We modified selected TPC-DS queries by introducing user-defined functions, with the goal of measuring the performance impact of stage separation (Section 6.2), and compared the results based on where the user code is located in the query plans.

We built the experiment on the 90% selectivity case on a non-partition column from Section 8.2, with the intent of forcing both *User* and *System* executors to be used within the same query. To account for the presence of both types of executors, we allocated a fixed 20 System executors and 20 User executors for Membrane. In typical production environments, AWS EMR customers have a choice between fixed-sized and auto-scaled clusters. Although providing an improved cost/performance ratio, dynamically scaling clusters introduces a variability into our benchmark that is difficult to account for. We have thus decided to pre-set the number of executors in order to eliminate the impact of cluster auto-scaling on query times. We still ensure a fair comparison with baseline with this adjustment, as at any given time, there can be at most 20 executors working on the query, all of either *System* or *User* type.

The user function used in this experiment is defined as a no-op scalar UDF *identity_udf()* that simply returns its input argument without any additional work. For each query involved in this experiment, we create two variants, differing by UDF placement:

- **UDF on SELECT**: we apply the UDF to one of the columns in the final SELECT list, to execute the user function close to the end of the query execution.
- **UDF on FILTER**: we introduce an additional filter *<column> = identity_udf(<column>)* to the WHERE clause of *store_sales* table, to execute the user function close to the scan of the corresponding table.

We pick 6 queries among all the TPC-DS queries based on their run time for this experiment: *q48* and *q70* to represent fast queries, *q47* and *q76* to represent medium-ranged run time queries, and *q29* and *q67* for slow queries. Figure 12 demonstrates the total execution time percentage of Membrane compared to the baseline. The *UDF on SELECT* case experiences the least overhead since all these queries already have at least one exchange in their plan, enabling Membrane to repurpose their final stage to be assigned to user executors and thus not needing an extra stage separation.

For the *UDF on FILTER* case, we expect a performance impact due to the unavoidable additional data exchange that happens immediately after the scan of the *store_sales* table. This overhead positively correlates to the size of the data that passes through the UDF. For example, *q76* incurs minimal overhead because less data is transferred (3.9Kib) due to other filters being evaluated prior to

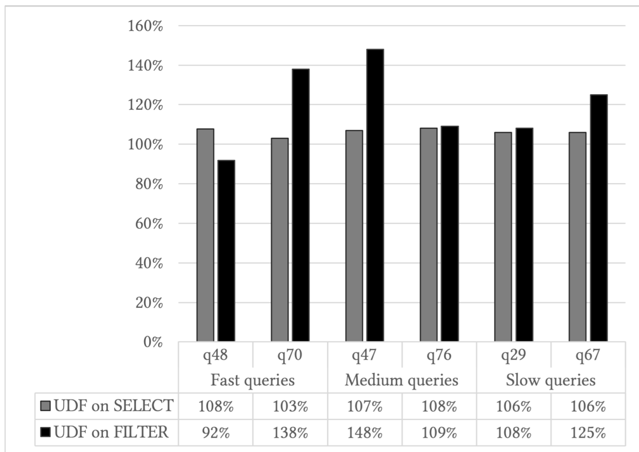


Figure 12: Total execution time percentage comparison of Membrane to baseline Spark for selected TPC-DS queries with data security filters on a non-partitioned column and 90% selectivity, grouped based on query run time.

the UDF, whereas other queries that transfer more data (5.3 GiB for *q67* and *q70*, 6.5 GiB for *q47*) experience higher overhead. There is one exceptional case, *q48*, that shows better performance on Membrane compared to baseline. When examined closer, we observed that, due to the security boundary operator separating the UDF filter from the partition filter, Membrane triggered an additional dynamic pruning on the *store_sales* table, reducing the size of the data that would be later exchanged for the UDF stage. Meanwhile in the baseline run, the UDF filter's presence preempted dynamic pruning from being applied and thus was evaluated against every single *store_sales* table entry without any opportunity for optimization, resulting in a more prominent performance impact. This is an opportunity for a future optimization work in open-source Spark.

9 RELATED WORK

Multiple publications recommend *query rewriting* for enforcing FGAC in database systems, with Stonebraker et al. [54] introducing it and LeFevre et al. [37] defining several ways of implementing it. The "Truman Model" [50] uses parameterized authorization views that replace the original table relation, providing each user with a "personal and restricted view of the complete database." Agrawal et al. [1] also propose query rewriting by generating a dynamic view and transforming the user's query into an equivalent query over the generated dynamic view. Oracle's Virtual Private Database [44] is a commercial implementation of FGAC using query rewriting as the main technique, with the policy defined as a function associated with each relation that returns a simple predicate added to the query's *WHERE* clause. Kabra et al. [35] discuss FGAC information leakage prevention and propose solutions like the generation of a safe query plan with respect to UDFs or unsafe functions that may cause an exception (a topic we also covered in Section 5.2). While some of the techniques presented above served as the foundation for Membrane, none of the literature we consulted addressed isolating the execution of user-provided code from the actual enforcement of FGAC policies. Due to Spark running the user program in the same memory space as the SQL compiler (where the query rewriting is being done) and running UDFs in the same processes that perform data access and filtering, additional work is needed to ensure safe enforcement of these policies, as presented in this paper.

Spark Connect [19] enables a remote client to connect to a Spark cluster using Dataset-compatible APIs by providing a frontend library that is separate from the actual Spark Driver and marshaling logical plans between the two. We found that a solution leveraging Spark Connect would have difficulties meeting our *Data Security* tenet. As of Spark version 3.5, it does not support for RDDs, UDTs or provide a mechanism to intercept or sandbox embedded user code. We decided to trade Spark Connect's potential benefits for the flexibility and security benefits of the User-System driver design described in Section 4. In future work, we may consider using Spark Connect's data model and marshaling as the foundation for the *Driver Transfer Marshaller* (Section 4.3), but that would be predicated on it supporting RDDs and UDTs.

In the Spark/Hadoop ecosystem, *GuardSpark++* [58] proposes a modification to Spark that uses query analysis and rewriting techniques to enforce purpose-aware fine-grained access controls. It assumes that the data management and sharing platforms are secure and fully trusted by both the data owners and users, hence the entire algorithm is inside the Spark SQL compiler. *Apache Sentry* [25] and *Apache Ranger* [24] are middlewares that can be used to define and apply FGAC rules, but they provide no mechanism for enforcing it in Spark. Membrane could be a used as a secure enforcement agent for policies defined in Apache Ranger.

10 CONCLUSION

Data Governance is an increasingly critical feature of modern database systems, with more and more customers opting to define granular security policies on their datasets and wanting to access them using popular tools like Apache Spark, with minimal impact to cost or performance. In this paper we presented Membrane: an innovative design that brings native fine-grained access controls into Apache Spark running on AWS EMR, built on the principle of sandboxing at the query stage boundary and designed to meet the high security bar for AWS services. The pillars that define it include securing the Spark SQL compiler and RDD execution pipeline from the Driver Program, rewriting query plans with logical security boundaries and employing container affinity techniques to physically separate user code execution from parts of the system that handle data governance rules enforcement.

ACKNOWLEDGMENTS

We are grateful to our leadership team and partners at AWS that made this project happen: Ganapathy (G2) Krishnamoorthy, Rick Sears, Julien Ellie and Gopinathan Kannan for helping bootstrap and supporting this project. We thank our teammates for their contributions: Amogh Jahagirdar, Armin Najafi, Chris Olson, Dhananjay Badaya, Fan Yang, Hansae Lee, Henry Mai, Jalpan Randeri, Jia Zhong, Mani Chandrasekar, Manu Khandelwal, Peter Slawski, Price Qian, Rajat Bhatt, Sangeet Lohariwala, Sam Zargar, Vineeth Sai Narajala, Yakov Shafranovich, Yasir Mukhtar, Yifan Zhao; their work was essential to implementing Membrane in AWS EMR.

REFERENCES

[1] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. 2005. Extending relational database systems to automatically enforce privacy policies. In *21st International Conference on Data Engineering (ICDE'05)*. 1013–1022. https://doi.org/10.1109/ICDE.2005.64

[2] Amazon. 2024. Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks. Accessed: January 22, 2024.

[3] Amazon. 2024. AWS Fargate. https://aws.amazon.com/fargate. Accessed: January 22, 2024.

[4] Amazon. 2024. Big Data Platform – Amazon EMR - AWS. https://aws.amazon.com/emr. Accessed: February 5, 2024.

[5] Amazon. 2024. Cloud Compute Capacity – Amazon EC2 - AWS. https://aws.amazon.com/ec2. Accessed: February 5, 2024.

[6] Amazon. 2024. Cloud Object Storage - Amazon S3 - AWS. https://aws.amazon.com/s3. Accessed: February 5, 2024.

[7] Amazon. 2024. Data Catalog and crawlers in AWS Glue. https://docs.aws.amazon.com/glue/latest/dg/catalog-and-crawler.html. Accessed: February 27, 2024.

[8] Amazon. 2024. Open-Source Big Data Analytics | Amazon EMR Serverless | Amazon Web Services. https://aws.amazon.com/emr/serverless. Accessed: February 5, 2024.

[9] Amazon. 2024. Secure Data Lake – AWS Lake Formation – AWS. https://aws.amazon.com/lake-formation. Accessed: February 5, 2024.

[10] Amazon. 2024. Serverless and Containers – Logical Separation on AWS. https://docs.aws.amazon.com/whitepapers/latest/logical-separation/serverless-and-containers.html. Accessed: January 29, 2024.

[11] Lyublena Antova, Amr El-Helw, Mohamed A. Soliman, Zhongxian Gu, Michalis Petropoulos, and Florian Waas. 2014. Optimizing queries over partitioned tables in MPP systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, 373–384. https://doi.org/10.1145/2588555.2595640

[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, 1383–1394. https://doi.org/10.1145/2723372.2742797

[13] Franziska Boenisch, Reinhard Munz, Marcel Tiepelt, Simon Hanisch, Christiane Kuhn, and Paul Francis. 2021. Side-Channel Attacks on Query-Based Data Anonymization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, 1254–1265. https://doi.org/10.1145/3460120.3484751

[14] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*. Association for Computing Machinery, 34–43. https://doi.org/10.1145/275487.275492

[15] Chen Dar, Moshik Hershcovitch, and Adam Morrison. 2023. RLS Side Channels: Investigating Leakage of Row-Level Security Protected Data Through Query Execution Time. *Proc. ACM Manag. Data* 1, 1, Article 89 (may 2023), 25 pages. https://doi.org/10.1145/3588943

[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[17] Nedim Dedić. and Clare Stanier. 2016. An Evaluation of the Challenges of Multilingualism in Data Warehouse Development. In *Proceedings of the 18th International Conference on Enterprise Information Systems - Volume 1: ICEIS*. INSTICC, SciTePress, 196–206. https://doi.org/10.5220/0005858401960206

[18] Apache Software Foundation. 2023. Spark Configuration – Stage Level Scheduling Overview. https://spark.apache.org/docs/3.5.0/configuration.html#stage-level-scheduling-overview. Accessed: January 29, 2024.

[19] Apache Software Foundation. 2023. Spark Connect. https://spark.apache.org/docs/3.5.0/spark-connect-overview.html. Accessed: February 14, 2024.

[20] Apache Software Foundation. 2024. Apache Hadoop YARN. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed: January 22, 2024.

[21] Apache Software Foundation. 2024. Apache Hive. https://hive.apache.org. Accessed: February 5, 2024.

[22] Apache Software Foundation. 2024. Apache Iceberg. https://iceberg.apache.org. Accessed: February 27, 2024.

[23] Apache Software Foundation. 2024. Apache Mesos. https://mesos.apache.org. Accessed: January 26, 2024.

[24] Apache Software Foundation. 2024. Apache Ranger. https://ranger.apache.org. Accessed: February 16, 2024.

[25] Apache Software Foundation. 2024. Apache Sentry. https://sentry.apache.org. Accessed: February 16, 2024.

[26] Apache Software Foundation. 2024. Apache Spark. https://spark.apache.org. Accessed: February 5, 2024.

[27] Apache Software Foundation. 2024. Scalar User Defined Aggregate Functions (UDAFs). https://spark.apache.org/docs/latest/sql-ref-functions-udf-aggregate.html. Accessed: January 29, 2024.

[28] Apache Software Foundation. 2024. Scalar User Defined Functions (UDFs). https://spark.apache.org/docs/latest/sql-ref-functions-udf-scalar.html. Accessed: January 29, 2024.

[29] Apache Software Foundation. 2024. Spark basicLogicalOperators.scala. https://github.com/apache/spark/blob/branch-3.5/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/plans/logical/basicLogicalOperators.scala#L316. Accessed: January 29, 2024.

[30] Apache Software Foundation. 2024. Spark Cluster Mode Overview. https://spark.apache.org/docs/latest/cluster-overview.html. Accessed: January 29, 2024.

[31] Apache Software Foundation. 2024. Spark QueryExecution.scala. https://github.com/apache/spark/blob/branch-3.5/sql/core/src/main/scala/org/apache/spark/sql/execution/QueryExecution.scala. Accessed: January 29, 2024.

[32] Apache Software Foundation. 2024. Spark UserDefinedType. https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/types/UserDefinedType.html. Accessed: January 22, 2024.

[33] Trino Software Foundation. 2024. Trino | Distributed SQL query engine for big data. https://trino.io. Accessed: February 5, 2024.

[34] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for {Fine-Grained} Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[35] Govind Kabra, Ravishankar Ramamurthy, and S. Sudarshan. 2006. Redundancy and information leakage in fine-grained access control. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1142473.1142489

[36] Kubernetes. 2024. Kubernetes Overview. https://kubernetes.io/docs/concepts/overview. Accessed: January 26, 2024.

[37] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. 2004. Limiting disclosure in hippocratic databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 108–119. https://dl.acm.org/doi/10.5555/1316689.1316701

[38] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, 418–429. https://doi.org/10.1145/3274694.3274720

[39] Microsoft. 2023. CLR Integration Code Access Security. https://learn.microsoft.com/en-us/sql/relational-databases/clr-integration/security/clr-integration-code-access-security. Accessed: February 27, 2024.

[40] Microsoft. 2024. Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs. Accessed: February 27, 2024.

[41] Sabyasachi Mitra and Sam Ransbotham. 2015. Information Disclosure and the Diffusion of Information Security Attacks. *Information Systems Research* 26, 3 (2015), 565–584. https://doi.org/10.1287/isre.2015.0587

[42] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 1049–1058. https://dl.acm.org/doi/10.5555/1182635.1164217

[43] Thomas Neumann. 2021. Evolution of a compiling query engine. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3207–3210. https://doi.org/10.14778/3476311.3476410

[44] Oracle. 2002. The Virtual Private Database in Oracle9iR2. https://www.cgisecurity.com/database/oracle/pdf/VPD9ir2twp.pdf.

[45] Oracle. 2020. Behind the scenes: How do lambda expressions really work in Java? https://blogs.oracle.com/javamagazine/post/behind-the-scenes-how-do-lambda-expressions-really-work-in-java. Accessed: January 26, 2024.

[46] Oracle. 2024. Java Object Serialization. https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html. Accessed: January 22, 2024.

[47] Hwee Hwa Pang, Hang Jun Lu, and Beng Chin Ooi. 1991. An efficient semantic query optimization algorithm. In *[1991] Proceedings. Seventh International Conference on Data Engineering*. 326–335. https://doi.org/10.1109/ICDE.1991.131480

[48] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *12th USENIX Security Symposium*. USENIX Association. https://www.usenix.org/conference/12th-usenix-security-symposium/preventing-privilege-escalation

[49] Python. 2024. Python object serialization. https://docs.python.org/3/library/pickle.html. Accessed: January 26, 2024.

[50] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. 2004. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 551–562. https://doi.org/10.1145/1007568.1007631

[51] Scala. 2024. Case Classes | Tour of Scala | Documentation. https://docs.scala-lang.org/tour/case-classes.html. Accessed: January 26, 2024.

[52] Shashi Shekhar, Jaideep Srivastava, and Soumitra Dutta. 1992. A formal model of trade-off between optimization and execution costs in semantic query optimization. *Data Knowledge Engineering* 8, 2 (1992), 131–151. https://doi.org/10.1016/0169-023X(92)90034-9

[53] Snyk. 2020. Serialization and deserialization in Java. https://snyk.io/blog/serialization-and-deserialization-in-java. Accessed: January 26, 2024.

[54] Michael Stonebraker and Eugene Wong. 1974. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 Annual Conference - Volume 1 (ACM '74)*. Association for Computing Machinery, New York, NY, USA, 180–186. https://doi.org/10.1145/800182.810400

[55] TPC. 2021. TPC Benchmark DS. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf. Accessed: February 29, 2024.

[56] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. Association for Computing Machinery, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[57] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[58] 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 *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20)*. Association for Computing Machinery, 582–596. https://doi.org/10.1145/3427228.3427640

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 15–28.