HITACHI

Hitachi Vantara

VSP One Object with Native Amazon S3 Tables

Implementation Guide

MK-SL-388-00
October 2025

© 2025 Hitachi Vantara LLC. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including copying and recording, or stored in a database or retrieval system for commercial purposes without the express written permission of Hitachi, Ltd., Hitachi Vantara, Ltd., or Hitachi Vantara LLC (collectively “Hitachi”). Licensee may make copies of the Materials provided that any such copy is: (i) created as an essential step in utilization of the Software as licensed and is used in no other manner; or (ii) used for archival purposes. Licensee may not make any other copies of the Materials. “Materials” mean text, data, photographs, graphics, audio, video and documents.

Hitachi reserves the right to make changes to this Material at any time without notice and assumes no responsibility for its use. The Materials contain the most current information available at the time of publication.

Some of the features described in the Materials might not be currently available. Refer to the most recent product announcement for information about feature and product availability, or contact Hitachi Vantara LLC at https://support.hitachivantara.com/en_us/contact-us.html.

Notice: Hitachi products and services can be ordered only under the terms and conditions of the applicable Hitachi agreements. The use of Hitachi products is governed by the terms of your agreements with Hitachi Vantara LLC.

By using this software, you agree that you are responsible for:

1. Acquiring the relevant consents as may be required under local privacy laws or otherwise from authorized employees and other individuals; and
2. Verifying that your data continues to be held, retrieved, deleted, or otherwise processed in accordance with relevant laws.

Notice on Export Controls. The technical data and technology inherent in this Document may be subject to U.S. export control laws, including the U.S. Export Administration Act and its associated regulations, and may be subject to export or import regulations in other countries. Reader agrees to comply strictly with all such regulations and acknowledges that Reader has the responsibility to obtain licenses to export, re-export, or import the Document and any Compliant Products.

Hitachi and Lumada are trademarks or registered trademarks of Hitachi, Ltd., in the United States and other countries.

AIX, DB2, DS6000, DS8000, Enterprise Storage Server, eServer, FICON, FlashCopy, GDPS, HyperSwap, IBM, IntelliMagic, IntelliMagic Vision, OS/390, PowerHA, PowerPC, S/390, System z9, System z10, Tivoli, z/OS, z9, z10, z13, z14, z15, z16, z17, z/VM, and z/VSE are registered trademarks or trademarks of International Business Machines Corporation.

Active Directory, ActiveX, Bing, Excel, Hyper-V, Internet Explorer, the Internet Explorer logo, Microsoft, Microsoft Edge, the Microsoft corporate logo, the Microsoft Edge logo, MS-DOS, Outlook, PowerPoint, SharePoint, Silverlight, SmartScreen, SQL Server, Visual Basic, Visual C++, Visual Studio, Windows, the Windows logo, Windows Azure, Windows PowerShell, Windows Server, the Windows start button, and Windows Vista are registered trademarks or trademarks of Microsoft Corporation. Microsoft product screen shots are reprinted with permission from Microsoft Corporation.

All other trademarks, service marks, and company names in this document or website are properties of their respective owners.

The open source content used in Hitachi Vantara products may be found within the Product documentation or you may request a copy of such information (including source code and/or modifications to the extent the license for any open source requires Hitachi make it available) by sending an email to OSS_licensing@hitachivantara.com.

Feedback

Hitachi Vantara welcomes your feedback. Please share your thoughts by sending an email message to doc.feedback@hitachivantara.com. To assist the routing of this message, use the paper number in the subject and the title of this white paper in the text.

Thank you!

Revision history

Changes	Date
Initial release	October 2025

Contents

Implementation Guide....4

- Hitachi VSP One Object Data Lakehouse Architecture....5
- Set Up VSP One Object Amazon S3 Tables....7
  - User Creation and Group Assignment....7
    - Add a User....7
    - Assign Credentials....8
  - S3 Credentials and Application Client Credentials....8
    - Generate S3 Credentials and Application Client Credentials....8
    - Create S3 Catalog Client Credentials for Spark and Trino....8
    - Create a Bucket, Namespace, and Table....9
- Integrate Apache Trino....14
- Integrate Apache Spark....18
- Automatic S3 Table Bucket and Table Maintenance....24
- Conclusion....29
- Appendix A: Detailed Test Environment Specifications....30
- Appendix B: Detailed Trino Docker Deployment....31
- Appendix C: Detailed Spark Cluster Docker Deployment....42
- Appendix D: AWS CLI Examples....51
- Appendix E: Python Examples....52
- Appendix F: Troubleshooting and Tips....55

Implementation Guide

This guide describes the setup and usage of the VSP One Object Native Amazon S3 Table feature. It demonstrates how to configure the platform and integrate it with Apache Spark and Apache Trino to ingest, manage, and query structured data. By following the steps outlined, readers will gain practical insights into building scalable, SQL-driven analytics workflows on Hitachi VSP One Object storage.

Introduction

VSP One Object is part of Hitachi Vantara's unified Virtual Storage Platform (VSP) One data plane, which integrates block, file, and object storage. As the enterprise-grade object storage component, VSP One Object delivers scalable, secure, and self-healing storage for diverse workloads, including backup, archiving, analytics, and edge computing. It supports both on-premises and hybrid cloud deployments with robust encryption, redundancy, and high-performance access.

The release of VSP One Object v3.1 marks a significant evolution in object storage, positioning it as a high-performance platform for structured analytics. With native support for Amazon S3 Tables, this capability enables seamless SQL querying on open-format data—eliminating the need for traditional ETL pipelines and unlocking transformative use cases across AI/ML, BI dashboards, and real-time insights.

Bridging Data Lakes and Warehouses

VSP One Object bridges the gap between traditional data lakes and data warehouses by simplifying lakehouse architecture. Users can query data in place by leveraging engines, such as Trino, Presto, or Apache Spark without moving data to external systems. This reduces complexity, accelerates time to insight, and lowers infrastructure costs.

Apache Iceberg Integration

The platform includes a built-in Iceberg catalog that is always consistent with Iceberg tables stored in VSP One Object. It is fully compliant with the Apache Iceberg REST API and can also be used to manage externally stored Iceberg data. Tables can be queried directly using Iceberg-compatible clients such as Trino, and Spark with Iceberg extensions.

Unlike do-it-yourself Iceberg deployments—where teams must provision catalogs, align S3 bucket structures, and schedule manual table maintenance—VSP One Object delivers these capabilities natively. Each S3 Table bucket includes built-in automation for essential maintenance tasks such as metadata cleanup, file compaction, and optimization. Administrators can simply define maintenance policies at the bucket level, ensuring predictable performance, lower costs, and reduced operational complexity.

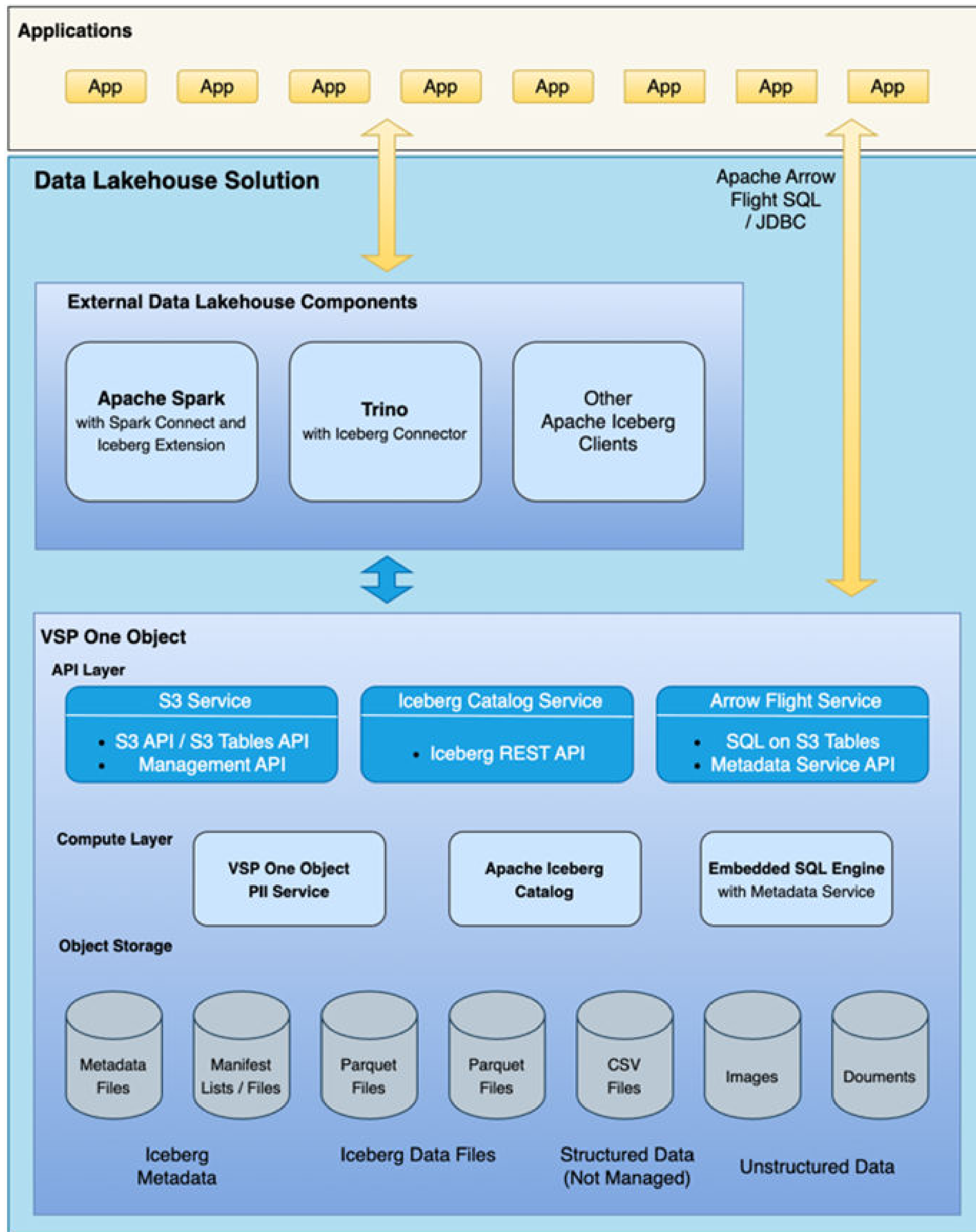
Hitachi VSP One Object Data Lakehouse Architecture

With the release of VSP One Object v3.1, Hitachi Vantara has redefined what enterprise object storage can do—transforming it into a high-performance platform for structured analytics. By integrating Amazon S3 Tables and a built-in Apache Iceberg REST catalog, VSP One Object now forms the foundation of a modern data lakehouse architecture that unifies storage, compute, and analytics.

In the following reference architecture, Apache Spark and Trino serve as the primary processing engines, operating directly on data stored in VSP One Object without the need for duplication, movement, or conversion. This enables organizations to simplify their data pipelines and accelerate insights, while maintaining flexibility, scalability, and strong governance controls.

- Apache Spark is used for batch ETL, machine learning workflows, and streaming ingestion, writing data into Iceberg tables that reside in VSP One Object.
- Trino, an MPP SQL query engine, provides low-latency SQL access to the same Iceberg tables, making it ideal for interactive analytics, BI dashboards, and federated query workloads across multiple sources.
- The shared Iceberg table format and central metadata catalog ensure ACID compliance, schema evolution, and multi-engine interoperability, all while storing data efficiently in the VSP One Object platform.

The following illustration shows the architecture overview.


Applications
App
App
App
App
App
App
App
App
Data Lakehouse Solution
Apache Arrow
Flight SQL
/ JDBC
External Data Lakehouse Components
Apache Spark
with Spark Connect and
Iceberg Extension
Trino
with Iceberg Connector
Other
Apache Iceberg
Clients
VSP One Object
API Layer
S3 Service
S3 API / S3 Tables API
Management API
Iceberg Catalog Service
Iceberg REST API
Arrow Flight Service
SQL on S3 Tables
Metadata Service API
Compute Layer
VSP One Object
PII Service
Apache Iceberg
Catalog
Embedded SQL Engine
with Metadata Service
Object Storage
Metadata
Files
Manifest
Lists / Files
Parquet
Files
Parquet
Files
CSV
Files
Images
Douments
Iceberg
Metadata
Iceberg Data Files
Structured Data
(Not Managed)
Unstructured Data

At a high level, this architecture is composed of the following components:

- Hitachi VSP One Object
  - Stores structured data in Iceberg table format using the native Amazon S3 Tables API
  - Hosts a built-in Iceberg REST catalog for metadata management
  - Automatic S3 Table Bucket and Table maintenance
- Apache Spark
  - Performs heavy data transformation, ETL, and streaming ingestion
  - Reads and writes Iceberg tables stored in VSP One Object
  - Leverages the embedded Iceberg REST catalog
- Trino
  - Runs interactive and federated SQL queries across Iceberg tables
  - Uses the same catalog and object store, enabling consistent data access
  - Scales horizontally to support concurrent BI and analytic workloads

This architecture enables data engineers and analysts to work in parallel on the same data foundation—Spark writing, Trino reading, and VSP One Object ensuring consistency and durability. By converging compute and storage around open standards such as Apache Iceberg and Amazon S3 APIs, this solution enables a vendor-neutral, cloud-ready, and engine-agnostic lakehouse—ideal for enterprise-scale data platforms.

Set Up VSP One Object Amazon S3 Tables

User creation and group assignment

This section walks through the foundational setup required to use VSP One Object Native Amazon S3 Table support. It includes user creation, group assignment, credential generation, and the creation of buckets, namespaces, and S3 Tables.

User Creation and Group Assignment

To manage S3 Tables effectively, users must be created and assigned to specific groups that grant the necessary permissions.

Add a User

Procedure

1. Open the **Keycloak** app.
2. Go to **Users** from the side panel.
3. Click **Add User** and fill out the required fields.
4. Click **Join Groups** and assign the new user to the following groups to enable S3 Table management:
   a. **S3 Bucket Management**.

   b. **Monitoring**.

5. Click **Create**. The user will appear in the **Username** column.
6. Select the new user and go to the **Credentials** tab.
7. Click **Set Password**, enter a password, and click **Save**.

   After creating the new user, you'll also need to set the user credentials.

Assign Credentials

After creating the new user, you'll need to set the user credentials.

Procedure

1. Click the user to view the user details then click the **Credentials** tab.
2. Click **Set Password** and enter the credentials password.
3. Click **Save** to finalize the password setup.

S3 Credentials and Application Client Credentials

Generate S3 Credentials and Application Client Credentials

S3 credentials (Access Key and Secret Key) are required to interact with the S3 gateway.

1. Log in as the new user.
2. Go to S3 > Buckets in Storage Manager.
3. Click Generate Credentials.
4. Confirm the action when prompted.
5. Copy and securely store the Access Key and Secret Key. These are only visible once.

Create S3 Catalog Client Credentials for Spark and Trino

Client catalog credentials are required when integrating applications such as Apache Spark or Apache Trino.

If your user does not have access to the Security Group in Keycloak, you'll need a security admin to help with this task.

Create the S3 catalog client credentials for client application access:

1. Open the Keycloak app.
2. Go to Clients from the side panel.
3. Click Create Client.
4. Select OpenID Connect as the Client Type.
5. Specify a Client ID: (example: `trino-client`).
6. Specify a Name: (example: `trino-client`).
7. Specify a Description.
8. Leave Always display in UI disabled.
9. Click Next.
10. Enable Client authentication.

11. Select Standard flow, Direct Access grants, Service accounts roles, and OAuth 2.0 Device Authorization Grant.
12. Click Next and click Save.

After the client has been created, do the following:

1. Click Client scopes under the new client's details.
2. Click Add client scope and add `s3catalog-scope`.
3. Click Add and select Default.

After adding the `s3catalog-scope` to the new client’s Client scopes, do the following:

1. Click the Credentials tab.
2. Note the Client Secret by clicking the eye icon.

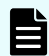
If you created a client for both Spark and for Trino you should have two new clients with their client secrets. You will use the client ID and client secrets for both Spark and Trino, as shown in the following example.

Client credentials
Client ID: `spark-client` Client Secret: `mXBr3PLyKBOJfTrhzUfkJw51kr3jCOpd`
Client ID: `trino-client` Client Secret: `pYDt4QMnLCPKeUshxWglKv62ms4kFJqr`

Create a Bucket, Namespace, and Table

Create an S3 Table bucket using the API

S3 Table buckets are the primary containers for table data. They can be created programmatically using tools such as the AWS S3 CLI or the `boto3` Python library.



Note: Before using the AWS CLI, make sure your credentials are configured in `~/.aws/credentials` and any configuration options are set in `~/.aws/config`. For more information, see the AWS CLI user guide: https://docs.aws.amazon.com/cli/v1/userguide/cli-configure-files.html.

Create an S3 Table bucket using the AWS CLI:

```
aws s3tables create-table-bucket --name <new-bucket-name> --region <region-name> --no-verify-ssl --endpoint https://s3.<region>.<system-name>.company.com
```

Example: If your bucket is in the region “site02” and your system is “100gvsp1o”, you can add your “airlinedata” as follows:

```
aws s3tables create-table-bucket --name airlinedata --region site02 --no-verify-ssl --endpoint https://s3.site02.100gvsp1o.company.com
```

Example: Use the `boto3` Python client to create a table bucket named `airlinedata`:

Note that the table ARN is being captured in this code snippet. The ARN will be important when creating namespaces and tables.

```
import boto3

endpoint_url = 'https://s3.site02.100gvsp1o.company.com'
aws_access_key_id = <your-access-key>
aws_secret_access_key = <your-secret-key>
region_name = 'site02'
verify = False
table_bucket = 'airlinedata' # We will use this for our S3 Table Bucket name

client = boto3.client(
    's3tables',  # Note we are using 's3tables' not 's3'
    endpoint_url=endpoint_url,
    aws_access_key_id=aws_access_key_id,
    aws_secret_access_key=aws_secret_access_key,
    region_name=region_name,
    verify=verify
)

resp = client.create_table_bucket(name=table_bucket)
ARN = resp['arn']
print(ARN)
arn:aws:s3tables:site02:726120222:bucket/airlinedata
```

Namespaces

Namespaces are used to organize S3 Tables within an S3 Table Bucket. At least one namespace is needed to create an S3 Table.

Obtain the bucket ARN

The S3 Table bucket ARN (Amazon Resource Name) is a unique identifier for an S3 Table bucket.

To create the namespace, the table bucket ARN is required. The bucket ARN is constructed as follows:

```
arn:aws:s3tables:<region>:< ownerAccountId>:bucket/<bucketName>
```

Note: We are using the `s3api` to capture the `ownerAccountId`, not the `s3tablesapi`.

```
aws s3api get-bucket-acl --bucket airlinedata --endpoint https://
s3.site02.100gvsp1o.company.com --no-verify-ssl
```

Example: Output from `get-bucket-acl` command:

```
{
    "Owner": {
```

```
        "DisplayName": "Arthur Dent",
        "ID": "726120222"
    },
    "Grants": [
        {
            "Grantee": {
                "DisplayName": " Arthur Dent",
                "ID": "726120222",
                "Type": "CanonicalUser"
            },
            "Permission": "FULL_CONTROL"
        }
    ]
}
```

Given the get-bucket-acl output, the bucket ARN can be constructed as:

```
arn:aws:s3tables:site02:726120222:bucket/airlinedata
```

Create the namespace

Create a namespace in a table bucket using the AWS CLI:

```
aws s3tables create-namespace --table-bucket-arn <table-bucket-ARN> --namespace
<namespace> --
region <region> --no-verify-ssl --endpoint https://s3.<region>.<system-
name>.company.com
```

Example: Create a namespace named “flightns” in table bucket “airlinedata.”

```
aws s3tables create-namespace --table-bucket-arn
arn:aws:s3tables:site02:726120222:bucket/airlinedata --namespace flightns –region
site02 --no-
verify-ssl --endpoint https://s3.site02.100gvsp1o.company.com
```

Example: List the new namespace:

```
aws s3tables list-namespaces --table-bucket-arn
arn:aws:s3tables:site02:726120222:bucket/airlinedata --region site02 --no-verify-ssl -
-endpoint
https://s3.site02.100gvsp1o.company.com
```

Example: Namespace listing:

```
{
    "namespaces": [
        {
            "namespace": [
                "flightns"
            ],
```

```
            "createdAt": "2025-08-07T16:49:34.614824+00:00",
            "createdBy": "TBD",
            "ownerAccountId": "726120222",
            "namespaceId": "01988570-1f9d-75d3-a4cd-261c515b82db",
            "tableBucketId": "1339970879"
        }
    ]
}
```

In Python, this process is much easier. When the table bucket is created using the `boto3` Python client, the table bucket ARN is included in the response. From the previous Python example, we simply obtained the ARN from the client `create_table_bucket` response:

```
resp = client.create_table_bucket(name=table_bucket)
ARN = resp['arn']
print(ARN)
arn:aws:s3tables:site02:726120222:bucket/airlinedata
```

The ARN output from the boto3 `create_table_bucket` function can then be passed into the `create_namespace` function:

Example: Create a table bucket namespace named ‘flightns’:

```
client.create_namespace(
    tableBucketARN=ARN,
    namespace=['flightns']
)
```

Create an S3 Table

After the bucket and namespace have been created, you can define an S3 Table using the AWS CLI or using the Python `boto3` client.

When creating an S3 Table you must provide a schema.

Example: Create an S3 Table named ‘flightdata’ in the namespace ‘flightns’ using the AWS CLI.

The following example shows the creation of a table with 5 columns. This is a shortened version of the airline data schema used in lab tests.

```
aws s3tables create-table --cli-input-json '{
    "tableBucketARN": "arn:aws:s3tables:site02:726120222:bucket/airlinedata",
    "namespace": " flightns'",
    "name": "flightdata",
    "format": "ICEBERG",
    "metadata": {
        "iceberg": {
            "schema": {
                "fields": [
                    {"name": "flight_id", "type": "string", "required": True},
                    {"name": "airline", "type": "string", "required": False},
```

```
                    {"name": "flight_number", "type": "int8", "required": False},
                    {"name": "arrival_delay", "type": " int8", "required": False},
                    {"name": "departure_delay", "type": " int8", "required": False},
                    ...
                ]
            }
        }
    }
}'  --region site02 --no-verify-ssl --endpoint https://s3.site02.100gvsp1o.company.com
```

Example: Create an S3 Table using the Python boto3 client.

This example shows the creation of a table with 5 columns. This is a shortened version of the airline data schema used in lab tests.

```
schema =  {
        "iceberg": {
            "schema": {
                "fields": [
                    {"name": "flight_id", "type": "string", "required": True},
                    {"name": "airline", "type": "string", "required": False},
                    {"name": "flight_number", "type": "int8", "required": False},
                    {"name": "arrival_delay", "type": " int8", "required": False},
                    {"name": "departure_delay", "type": " int8", "required": False},
                    ...
                ]
            }
        }
    }

resp = client.create_table(
    tableBucketARN=ARN,
    namespace=namespace,
    name="flightdata",
    format="ICEBERG",
    metadata=schema
)
tableARN = resp['tableARN'] # capture and display the S3 table ARN
print(tableARN)
arn:aws:s3tables:site02:726120222:bucket/airlinedata/table/flightdata
```

Integrate Apache Trino

Prerequisites

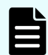
Before configuring Trino for access to the VSP One Object store, make sure the following requirements have been met. This section covers the high-level configuration options for Apache Trino. This section assumes that Trino is already installed and running in your environment. Appendix B (on page 31) covers in detail on how to deploy a small Trino cluster using Docker and docker-compose.

System requirements

- Network access to VSP One Object endpoints
- Bucket Access and Secret Keys have been created
- Client credentials for Trino have been created
- Table bucket, Namespace, and S3 Table has been created

SSL certificate setup

In addition to the Bucket Access Key, Secret Access Key and client credentials, the Keycloak and S3 CA certificates are required for the Trino configuration. These Keycloak and S3 CA certificates will be added to the java keystore in your Trino deployment environment. Adding the self-signed certs to Trino will prevent SSL verification errors.



Note: These steps are only necessary if VSP One Object uses self-signed certificates.

To export the Keycloak CA certificate:

```
openssl s_client -connect admin.gms.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/kc.pem
```

To export the S3 CA certificate:

```
openssl s_client -connect s3.site02.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/s3.pem
```

High-Level Trino configuration

Import the Keycloak and S3 certificates

Copy the kc.pem and s3.pem to the Trino coordinator and worker nodes. In the example below, the pem files have been copied to the /etc/trino/custom-ca/ folder.

On the coordinator and the worker nodes run the java keytool tool to import the Keycloak and S3 CA certs:

```
keytool -import -alias s3 -keystore $JAVA_HOME/lib/security/cacerts -file /etc/trino/
custom-ca/s3.pem -storepass changeit -noprompt
```

```
keytool -import -alias kc -keystore $JAVA_HOME/lib/security/cacerts -file /etc/trino/
custom-ca/kc.pem -storepass changeit -noprompt
```

Verify that the certs were added to the java keystore:

```
keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit | grep s3
keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit | grep kc
```

After importing the Keycloak and S3 certs into the java keystore, create a catalog properties file for the new bucket.

Trino catalog properties

- The “iceberg.rest-catalog.uri” property points to the VSP One Object catalog endpoint.
  - The endpoint has the form: https://lakekeeper.s3.<region>.<system-name>.company.com/catalog.
- The “iceberg.rest-catalog.oauth2.credential” will be the S3 catalog clients generated in the previous steps in the format <client-name>:<client-secret>.
- The “iceberg.rest-catalog.oauth2.server-uri” is the VSP One Object Keycloak endpoint.
  - The endpoint has the form: https://admin.gms.<system-name>.company.com/ui/auth/realms/vsp-object/protocol/openid-connect/token.
- The “iceberg.rest-catalog.warehouse” property is where the bucket is specified.
- The “s3.aws-access-key” and “s3.aws-secret-key” are the bucket access key and secret keys generated in the previous steps.
- The “s3.region” property specifies the regions. In our examples our region is “site02”.

After the catalog properties have been defined, copy the catalog properties file to the folder `/etc/trino/catalog/` on the coordinator node and each worker node.

Example: Trino catalog configuration file, `airlinedata.properties`.

```
connector.name=iceberg
fs.native-s3.enabled=true
iceberg.catalog.type=rest
iceberg.file-format=PARQUET
iceberg.rest-catalog.uri=https://lakekeeper.s3.site02.100gvsp1o.company.com/catalog
iceberg.rest-catalog.oauth2.scope=s3catalog-scope
iceberg.rest-catalog.security=OAUTH2
iceberg.rest-catalog.oauth2.credential=trino-client:mXBr3PLyKBOJfTrhzUfkJw51kr3jCOpd
iceberg.rest-catalog.oauth2.server-uri=https://admin.gms.100gvsp1o.company.com/ui/
auth/realms/vsp-object/protocol/openid-connect/token
iceberg.rest-catalog.vended-credentials-enabled=false
```

```
iceberg.rest-catalog.warehouse=airlinedata
s3.aws-access-key=AKIA7X9B3vQWmTzKJrLp
s3.aws-secret-key=ZTk1NzYxMmEtYjYxMi00NjQyLTkzNzEtYzY2YjQw
s3.connection-max-idle-time=5m
s3.connection-ttl=1h
s3.endpoint=https://s3.site02.100gvsp1o.company.com
s3.path-style-access=true
s3.region=site02
s3.socket-connect-timeout=10s
s3.socket-read-timeout=10m
s3.tcp-keep-alive=true
```

To verify Trino access to VSP One Object S3 Table, use the Trino cli to access the previously created S3 table.

Using the Trino cli, verify that Trino can access the previously created S3 table.

You can download the Trino cli from here: https://trino.io/docs/current/client/cli.html

Example: Verify that Trino can display the defined catalogs.

```
trino> show catalogs;
   Catalog
--------------
 airlinedata
 system
 warehouse000
(3 rows)

Query 20250814_205855_17946_95avu, FINISHED, 3 nodes
Splits: 53 total, 53 done (100.00%)
0.07 [0 rows, 0B] [0 rows/s, 0B/s]
```

Example: Verify that Trino can access S3 Tables stored on VSP One Object

The following example shows the full flightdata schema used in lab tests.

```
trino> USE airlinedata.flightns;
USE
trino:flightns> DESCRIBE flightdata;
          Column          |     Type     | Extra | Comment
--------------------------+--------------+-------+---------
 flight_id                | varchar      |       |
 airline                  | varchar      |       |
 flight_number            | varchar      |       |
 aircraft_type            | varchar      |       |
 departure_airport        | varchar      |       |
 arrival_airport          | varchar      |       |
 scheduled_departure_time | timestamp(6) |       |
 scheduled_arrival_time   | timestamp(6) |       |
 actual_departure_time    | timestamp(6) |       |
 actual_arrival_time      | timestamp(6) |       |
```

```
 departure_delay          | integer      |       |
 arrival_delay            | integer      |       |
 passenger_count          | integer      |       |
 ticket_price             | decimal(8,2) |       |
 flight_duration_minutes  | integer      |       |
 distance_miles           | integer      |       |
 status                   | varchar      |       |
 gate                     | varchar      |       |
 pilot_id                 | varchar      |       |
 crew_count               | integer      |       |
 created_at               | timestamp(6) |       |
 updated_at               | timestamp(6) |       |
 data_source              | varchar      |       |
 record_version           | integer      |       |
(24 rows)

Query 20250814_211123_17950_95avu, FINISHED, 3 nodes
Splits: 53 total, 53 done (100.00%)
0.22 [24 rows, 1.85KiB] [107 rows/s, 8.3KiB/s]
```

After populating the flightdata S3 table, the data can be queried using the Trino cli or using the Trino python client.

Example: Query data in the S3 Table.

```
trino:flightns> SELECT airline, flight_number, departure_delay, arrival_delay FROM
flightdata ORDER BY arrival_delay DESC LIMIT 10;
      airline      | flight_number | departure_delay | arrival_delay
-------------------+---------------+-----------------+---------------
 Delta Air Lines   | DL1515        |             505 |           490
 Alaska Airlines   | AS4083        |             493 |           484
 American Airlines | AA877         |             490 |           482
 United Airlines   | UA7447        |             488 |           480
 United Airlines   | UA8300        |             432 |           462
 Spirit Airlines   | NK784         |             437 |           458
 United Airlines   | UA6024        |             447 |           445
 JetBlue Airways   | B66159        |             435 |           443
 JetBlue Airways   | B69403        |             401 |           408
 United Airlines   | UA7011        |             390 |           401
(10 rows)

Query 20250814_211746_17952_95avu, FINISHED, 2 nodes
Splits: 1,500 total, 1,500 done (100.00%)
2.28 [1.47K rows, 10.5MiB] [644 rows/s, 4.63MiB/s]
```

Integrate Apache Spark

Prerequisites

Before configuring Spark to access the VSP One Object store, make sure the following requirements are met. This section covers the configuration options for a driver-only (local) PySpark + Iceberg deployment. For details on using VSP One Object S3 Tables with a small Spark cluster deployed with Docker, see Appendix C (on page 42).

System requirements

- A Linux environment to serve as the Spark driver node (tested on Ubuntu 24.04.2 LTS)
- A working directory (for example: `/home/user/pyspark`)
- Java 17 (required for Spark 3.4.x)
  - JAVA_HOME has been set `export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64`
- Python 3.11 or 3.12
  - A Python virtual environment has been created and activated:
    - `python3 -m venv ~/pyspark/.venv`
    - `source ~/pyspark/.venv/bin/activate`
- Network access to VSP One Object endpoints
- Bucket Access and Secret Keys have been created
- Client credentials for Trino have been created
- A t able bucket, namespace, and S3 Table have been created

SSL Certificate setup

In addition to the Bucket Access Key, Secret Access Key, and client credentials, the Keycloak and S3 CA certificates are required for the Spark configuration. These Keycloak and S3 CA certificates must be added to the java keystore where Spark is running. Adding the self-signed certificates to the Java keystore prevents SSL verification errors.

Note: These steps are only necessary if VSP One Object uses self-signed certificates.

To export the Keycloak CA certificate:

```
openssl s_client -connect admin.gms.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/kc.pem
```

Export the S3 CA certificate:

```
openssl s_client -connect s3.site02.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/s3.pem
```

High-level Spark configuration

Import the Keycloak and S3 certificates

Copy the kc.pem and s3.pem certificates to the Spark driver nodes. Since this example is for standalone mode, place the PEM files in a directory accessible to the JVM—such as `/home/user/pyspark/pem/`—so they can be imported into the Java keystore for secure communication with VSP One Object endpoints.

On the Spark driver node run the java keytool tool to import the Keycloak and S3 CA certs. This will add them system wide:

```
sudo keytool -import -alias s3 -keystore $JAVA_HOME/lib/security/cacerts -file /home/
user/pyspark/pem/s3.pem -storepass changeit -noprompt
```

```
sudo keytool -import -alias kc -keystore $JAVA_HOME/lib/security/cacerts -file /home/
user/pyspark/pem/kc.pem -storepass changeit -noprompt
```

Verify that the certs were added to the java keystore:

```
sudo keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit |
grep s3
sudo keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit |
grep kc
```

Note: If you don't have root or sudo access to your system, the S3 and Keycloak certs can be added locally in a home directory or other location.

```
keytool -import -alias s3 -keystore cacerts -file /home/user/pyspark/pem/s3.pem -
storepass changeit -noprompt
```

```
keytool -import -alias kc -keystore cacerts -file /home/user/pyspark/pem/kc.pem -
storepass changeit -noprompt
```

Verify that the certs were added to the local keystore (that is, **:**`/home/user/pyspark/cacerts`)

```
keytool -list -keystore cacerts -storepass changeit | grep s3
keytool -list -keystore cacerts -storepass changeit | grep kc
```

After importing the Keycloak and S3 certs into the java keystore, export the environment variable to that Spark and Iceberg will have access to the keystore:

Example: System Wide

```
export JAVA_TOOL_OPTIONS="-Djavax.net.ssl.trustStore=$JAVA_HOME/lib/security/cacerts -
Djavax.net.ssl.trustStorePassword=changeit"
```

Example: Local

```
export JAVA_TOOL_OPTIONS="-Djavax.net.ssl.trustStore=/home/user/pyspark/cacerts -
Djavax.net.ssl.trustStorePassword=changeit"
```

Download the Iceberg JARs

To use VSP One Object S3 tables with Spark, it will be necessary to download the Iceberg JARs as Spark does not include native support for Iceberg by default. These JARs enable Spark to read, write, and manage VSP One Object S3 tables, and to connect with the VSP One Object REST catalog.

Create a directory to contain the Iceberg JARs

```
mkdir -p /home/user/pyspark/opt/iceberg-jars
cd /home/user/pyspark/opt/iceberg-jars
```

Download the Iceberg Spark runtime and Iceberg AWS Bundle.

Note: The version numbers: Iceberg Spark Runtime 3.4 with Scala 2.12 and Iceberg AWS Bundle 1.6.1.

```
# Iceberg runtime for Spark 3.4 + Scala 2.12
wget https://repo1.maven.org/maven2/org/apache/iceberg/iceberg-spark-runtime-
3.4_2.12/1.6.1/iceberg-spark-runtime-3.4_2.12-1.6.1.jar

# Iceberg AWS bundle 1.6.1
wget https://repo1.maven.org/maven2/org/apache/iceberg/iceberg-aws-bundle/1.6.1/
iceberg-aws-bundle-1.6.1.jar
```

Install PySpark 3.4.2

Activate the python virtual environment and install PySpark 3.4.2.

```
source ~/pyspark/.venv/bin/activate
pip install pyspark==3.4.2
```

Build the Spark Session (Python / Notebook)

```
import os
import random
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

os.environ['JAVA_TOOL_OPTIONS'] = '-Djavax.net.ssl.trustStore=/home/user/pyspark/
cacerts -Djavax.net.ssl.trustStorePassword=changeit'

catalog_name = "airlinedata"
namespace_name = "flightns"
table_name = "flightdata"
```

```
s3_endpoint = "https://s3.site02.100gvsp1o.company.com"
lakekeeper_url = "https://lakekeeper.s3.site02.100gvsp1o.company.com/catalog"
token_url = "https://admin.gms.100gvsp1o.company.com/ui/auth/realms/vsp-object/
protocol/openid-connect/token"

data = {
    "grant_type": "client_credentials",
    "client_id": "spark-client",
    "client_secret": "mXBr3PLyKBOJfTrhzUfkJw51kr3jCOpd",
}

# Get Spark version for JAR compatibility
spark_version = pyspark.__version__
spark_min_version = ".".join(spark_version.split(".")[:2])  # e.g., "3.4.2"
iceberg_version = "1.6.1"

jar_dir = "/home/user/pyspark/opt/iceberg-jars"  # wherever you saved them
iceberg_runtime = f"{jar_dir}/iceberg-spark-runtime-3.4_2.12-1.6.1.jar"
iceberg_aws     = f"{jar_dir}/iceberg-aws-bundle-1.6.1.jar"

# Build Spark session with Iceberg configuration
spark = (
    SparkSession.builder
    .appName("iceberg-catalog-access")
    .master("local[*]")
    .config("spark.jars", f"{iceberg_runtime},{iceberg_aws}")

    # Enable Iceberg extensions
    .config("spark.sql.extensions",
"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions")

    # Catalog configuration
    .config(f"spark.sql.catalog.{catalog_name}",
"org.apache.iceberg.spark.SparkCatalog")
    .config(f"spark.sql.catalog.{catalog_name}.uri", lakekeeper_url)
    .config(f"spark.sql.catalog.{catalog_name}.warehouse", catalog_name)
    .config(f"spark.sql.catalog.{catalog_name}.type", "rest")
    .config(f"spark.sql.catalog.{catalog_name}.scope", "s3catalog-scope")

    # Authentication settings
    .config(f"spark.sql.catalog.{catalog_name}.credential", f"{data["client_id"]}:
{data["client_secret"]}")
    .config(f"spark.sql.catalog.{catalog_name}.grant-type", data["grant_type"])
    .config(f"spark.sql.catalog.{catalog_name}.oauth2-server-uri", token_url)
    .config(f"spark.sql.catalog.{catalog_name}.header.X-Iceberg-Access-Delegation",
"vendedcredentials")

    # SSL trust store configuration
    .config("spark.driver.extraJavaOptions", "-Djavax.net.ssl.trustStore=cacerts -
Djavax.net.ssl.trustStorePassword=changeit")
    .config("spark.executor.extraJavaOptions", "-Djavax.net.ssl.trustStore=cacerts -
```

```
Djavax.net.ssl.trustStorePassword=changeit")
    .getOrCreate()
)
```

Verify the Spark Session (Python / Notebook)

From a notebook or script, verify that the spark session can query data stored in the S3 Tables.

Example: Display the catalogs.

Note: Spark will not display the actual catalog names, but instead display “spark_catalog”. This is a known issue with Spark.

```
spark.sql("SHOW NAMESPACES IN airlinedata").show(truncate=False)

+-------------+
|catalog      |
+-------------+
|spark_catalog|
+-------------+
```

Example: Display the namespaces.

```
spark.sql("SHOW NAMESPACES IN airlinedata").show(truncate=False)

+-------------+
|catalog      |
+-------------+
|spark_catalog|
+-------------+
```

Example: Display the tables in the namespace.

```
spark.sql("SHOW TABLES IN airlinedata.flightns").show(truncate=False)

+---------+----------+-----------+
|namespace|tableName |isTemporary|
+---------+----------+-----------+
|flightns |flightdata|false      |
+---------+----------+-----------+
```

Example: Describe the flightdata table.

```
spark.sql("DESCRIBE TABLE airlinedata.flightns.flightdata").show(truncate=False)

+-----------------------+-------------+-------+
|col_name               |data_type    |comment|
+-----------------------+-------------+-------+
|flight_id              |string       |null   |
```

```
|airline                |string       |null   |
|flight_number          |string       |null   |
|aircraft_type          |string       |null   |
|departure_airport      |string       |null   |
|arrival_airport        |string       |null   |
|scheduled_departure_time|timestamp_ntz|null   |
|scheduled_arrival_time  |timestamp_ntz|null   |
|actual_departure_time   |timestamp_ntz|null   |
|actual_arrival_time     |timestamp_ntz|null   |
|departure_delay        |int          |null   |
|arrival_delay          |int          |null   |
|passenger_count        |int          |null   |
|ticket_price           |decimal(8,2) |null   |
|flight_duration_minutes |int          |null   |
|distance_miles         |int          |null   |
|status                 |string       |null   |
|gate                   |string       |null   |
|pilot_id               |string       |null   |
|crew_count             |int          |null   |
+------------------------+-------------+-------+
only showing top 20 rows
```

After populating the flightdata S3 table, the data can be queried using the newly created spark session.

Example: Query the flightdata table.

```
spark.conf.set("spark.sql.defaultCatalog", catalog_name)

query = f"SELECT airline, flight_number, departure_delay, arrival_delay FROM
{namespace_name}.{table_name} ORDER BY arrival_delay DESC LIMIT 10;"

spark.sql(query).show()

+-----------------+-------------+---------------+-------------+
|          airline|flight_number|departure_delay|arrival_delay|
+-----------------+-------------+---------------+-------------+
|  Delta Air Lines|       DL1515|            505|          490|
|  Alaska Airlines|       AS4083|            493|          484|
|American Airlines|        AA877|            490|          482|
|  United Airlines|       UA7447|            488|          480|
|  United Airlines|       UA8300|            432|          462|
|  Spirit Airlines|        NK784|            437|          458|
|  United Airlines|       UA6024|            447|          445|
|  JetBlue Airways|       B66159|            435|          443|
|  JetBlue Airways|       B69403|            401|          408|
|  United Airlines|       UA7011|            390|          401|
+-----------------+-------------+---------------+-------------+
```

Automatic S3 Table Bucket and Table Maintenance

Maintaining Iceberg tables is critical for performance and cost efficiency. Traditionally, this requires manual jobs for compaction, snapshot cleanup, and metadata optimization. With VSP One Object, these operations are embedded directly into the platform. Administrators can define maintenance policies at the bucket or table level, ensuring tasks run automatically on schedule. This simplifies operations, reduces overhead, and keeps data consistently optimized without extra tooling.

This section walks through how to set up and configure these automatic maintenance operations for both S3 Table buckets and individual tables.

S3 Table Bucket Maintenance: Unreferenced File Removal

Automatically remove unreferenced files and non-current data versions to reduce storage costs and maintain clean metadata.

Example: Configure unreferenced file removal.

```
aws s3tables put-table-bucket-maintenance-configuration \
  --table-bucket-arn arn:aws:s3tables:site02:726120222:bucket/airlinedata \
  --type icebergUnreferencedFileRemoval \
  --value \
'{
    "status":"enabled",
    "settings":{
        "icebergUnreferencedFileRemoval":{
            "unreferencedDays":4,
            "nonCurrentDays":10
        }
    }
}'
```

Example: Configure unreferenced file removal with python boto3 client.

```
config =
{"status":"enabled","settings":{"icebergUnreferencedFileRemoval":
{"unreferencedDays":4,"nonCurrentDays":10}}}

client.put_table_bucket_maintenance_configuration(
    tableBucketARN=ARN,
    type='icebergUnreferencedFileRemoval',
    value=config
)
```

Example: Verify unreferenced file removal.

```
aws s3tables get-table-bucket-maintenance-configuration \
--table-bucket-arn arn:aws:s3tables:site02:726120222:bucket/airlinedata

#### Output ####
```

```
{
    "tableBucketARN": "arn:aws:s3tables:site02:726120222:bucket/airlinedata",
    "configuration": {
        "icebergUnreferencedFileRemoval": {
            "status": "enabled",
            "settings": {
                "icebergUnreferencedFileRemoval": {
                    "unreferencedDays": 4,
                    "nonCurrentDays": 10
                }
            }
        }
    }
}
```

Example: Verify unreferenced file removal using python boto3 client.

```
client.get_table_bucket_maintenance_configuration(tableBucketARN=ARN)

#### Output ####

{'ResponseMetadata': {'HTTPStatusCode': 200,
  'HTTPHeaders': {'date': 'Thu, 28 Aug 2025 23:06:34 GMT',
   'server': 'uvicorn',
   'content-length': '228',
   'content-type': 'application/json'},
  'RetryAttempts': 0},
 'tableBucketARN': 'arn:aws:s3tables:site02:726120222:bucket/tpcds',
 'configuration': {'icebergUnreferencedFileRemoval': {'status': 'enabled',
   'settings': {'icebergUnreferencedFileRemoval': {'unreferencedDays': 4,
     'nonCurrentDays': 10}}}}}
```

S3 Table Maintenance: Table Compaction

Target file size

Set the target file size for compaction (between 64MB and 512MB):

```
aws s3tables put-table-maintenance-configuration \
  --table-bucket-arn arn:aws:s3tables:site02:726120222:bucket/airlinedata \
  --type icebergCompaction \
  --namespace flightns \
  --name flightdata \
  --value \
'{
    "status":"enabled",
    "settings":{
        "icebergCompaction":{
            "targetFileSizeMB":256
        }
```

```
 }'

client.put_table_maintenance_configuration(
    tableBucketARN=ARN,
    namespace=namespace,
    name=table_name,
    type="icebergCompaction",
    value=config
)
```

Compaction Strategy

“binpack”, is the current supported strategy on VSP One Object:

Set binpack as the compaction strategy:

```
aws s3tables put-table-maintenance-configuration \
   --table-bucket-arn arn:aws:s3tables:site02:726120222:bucket/airlinedata \
   --type icebergCompaction \
   --namespace flightns \
   --name flightdata \
   --value \
'{
    "status":"enabled",
    "settings":{
        "icebergCompaction":{
            "strategy":"binpack"
        }
    }
}'
```

Example: Set the table compaction strategy using the python boto3 client:

```
config = {
    "status":"enabled",
    "settings":{
        "icebergCompaction":{
            "strategy":"binpack"
        }
    }
}

client.put_table_maintenance_configuration(
    tableBucketARN=ARN,
    namespace=namespace,
    name=table_name,
    type="icebergCompaction",
    value=config
)
```

Disable Compaction

Disable compaction for a specific table:

```
aws s3tables put-table-maintenance-configuration \
   --table-bucket-arn arn:aws:s3tables:site02:726120222:bucket/airlinedata \
   --type icebergCompaction \
   --namespace flightns \
   --name flightdata \
   --value \
''{
    "status":"disabled",
    "settings":{
        "icebergCompaction":{
            "targetFileSizeMB":256
        }
 }''
```

Disable compaction using the python boto3 client:

```
config = {
        "status": "disabled",
        "settings": {
            "icebergCompaction": {
                "targetFileSizeMB": 256
            }
        }
    }

client.put_table_maintenance_configuration(
    tableBucketARN=ARN,
    namespace=namespace,
    name=table_name,
    type="icebergCompaction",
    value=config
)
```

S3 Table maintenance: Snapshot Management

Snapshot cleanup is essential for controlling metadata size and ensuring query performance. VSP One Object allows administrators to configure snapshot retention policies.

Enable Snapshot Management

Set minimum snapshots to retain and maximum snapshot age:

Enable snapshot management:

```
aws s3tables put-table-maintenance-configuration \
--table-bucket-arn arn:aws:s3tables:site02:726120222:bucket/airlinedata \
--namespace flightns \
--name flightdata \
```

```
--type icebergSnapshotManagement \
--value \
'{
    "status":"enabled",
    "settings":{
        "icebergSnapshotManagement":{
            "minSnapshotsToKeep":10,
            "maxSnapshotAgeHours":2500
        }
    }
}
'
```

Enable snapshot management using the python boto3 client:

```
config = {
    "status":"enabled",
    "settings":{
        "icebergSnapshotManagement":{
            "minSnapshotsToKeep":10,
            "maxSnapshotAgeHours":2500
            }
        }
    }


client.put_table_maintenance_configuration(
    tableBucketARN=ARN,
    namespace=namespace,
    name=table_name,
    type='icebergSnapshotManagement',
    value=config
)
```

Disable Snapshot Management

Disable snapshot cleanup:

```
aws s3tables put-table-maintenance-configuration \
--table-bucket-arn arn:aws:s3tables:site02:726120222:bucket/airlinedata \
--namespace flightns \
--name flightdata \
--type icebergSnapshotManagement \
--value \
'{
    "status":"disabled",
    "settings":{
        "icebergSnapshotManagement":{
            "minSnapshotsToKeep":10,
            "maxSnapshotAgeHours":2500
        }
```

```
    }
}
'
```

Disable snapshot management using the python boto3 client.

```
config = {
    "status":"disabled",
    "settings":{
        "icebergSnapshotManagement":{
            "minSnapshotsToKeep":1,
            "maxSnapshotAgeHours":120
        }
    }
}

client.put_table_maintenance_configuration(
    tableBucketARN=ARN,
    namespace=namespace,
    name=table_name,
    type='icebergSnapshotManagement',
    value=config
)
```

These built-in maintenance features allow VSP One Object to deliver a fully automated, production-grade Iceberg experience. By reducing the operational burden of manual table upkeep, administrators can enable scalable and cost-efficient lakehouse deployments.

Conclusion

The release of VSP One Object with Native AWS S3 Tables marks a significant advancement in the evolution of enterprise object storage, transforming it into a high-performance platform for structured analytics. The platform's Native Amazon S3 Table feature, combined with a built-in Apache Iceberg REST catalog and support for open standards, enables organizations to build modern data lakehouse architectures without the need for legacy ETL pipelines or complex infrastructure.

By integrating seamlessly with Apache Spark and Apache Trino, VSP One Object allows users to ingest, manage, and query structured data in place, accelerating time to insight and simplifying analytics pipelines. Apache Spark is leveraged for batch ETL, streaming, and ML workloads, while Trino delivers fast, federated SQL queries across Iceberg tables—all operating on a shared, consistent data layer backed by object storage.

The architecture described in this paper demonstrates how Spark and Trino can work in parallel, powered by the Iceberg table format and a shared REST metadata catalog. This enables high concurrency, ACID compliance, and engine interoperability. The result is an engine-agnostic lakehouse that supports real-time analytics, AI/ML pipelines, and self-service BI—all deployed on-premises or in the cloud using VSP One Object's S3 Table compatible APIs.

In short, VSP One Object delivers a robust, scalable foundation for modern analytics workflows. It bridges the gap between data lakes and data warehouses, enabling enterprises to unlock the full potential of their object data while reducing infrastructure complexity and operational costs. Whether used for AI/ML, business intelligence, archival, or real-time decision support, VSP One Object provides the performance, flexibility, and openness required to support the next generation of data-driven applications.

Appendix A: Detailed Test Environment Specifications

The following specifications outline the test environment that this architecture was developed on. For production use cases, a larger system consisting of multiple nodes running a container orchestration platform such as upstream Kubernetes or a more opinionated management platform such as RedHat OpenShift is recommended.

Hitachi VSP One Object

Component	Specification
VSP One Object	Version 3.1.0.2025-07-25.07201-505f26bc

Server specifications

Component	Specification
CPU	48 Cores (Dual 24 CPUs x Intel(R) Xeon(R) Silver 4310 CPU @ 2.10GHz)
CPU Hyperthreading	Enabled
Memory	128 GB
Network	1 Gbps
Internal Disk Capacity	2.18 TB
Operating System	VMware ESXi 8.0U3

VM specifications

Component	Specification
vCPU	16
Memory	32 GB
vNICs	2

Component	Specification
vDisk Capacity	500 GB
Operating System	Ubuntu 24.04.2
Container Management	Docker version 27.5.1, build 27.5.1-0ubuntu3~24.04.2
Docker Compose	docker-compose version 1.29.2, build unknown

Software specifications

Software	Version
Linux	Ubuntu 24.04.2
Java	OpenJDK 17.0.16 (build 17.0.16+8-Ubuntu-0ubuntu124.04.1)
Python	3.12.3
Spark/PySpark	3.4.2
Spark Container	quay.io/okdp/spark-py:spark-3.4.2-python-3.11-scala-2.12-java-17-2025-05-20-1.0.1
Iceberg Spark Runtime	iceberg-spark-runtime-3.4_2.12-1.6.1.jar
Iceberg AWS Bundle	iceberg-aws-bundle-1.6.1.jar
Trino	476
Trino Container	trinodb/trino:latest
Apache Superset	apache/superset:latest
JupyterLab	Version 4.4.5

Appendix B: Detailed Trino Docker Deployment

The following describes how to build and deploy a Trino cluster with Apache Superset using Docker and Docker Compose for use with a Hitachi VSP One Object store with S3 Table support.

Before installing and configuring Trino, make sure the following requirements have been met:

- Linux environment (tested on Ubuntu 24.04.2 LTS)
- Docker and Docker Compose installed
  - Tested on docker version 27.5.1, build 27.5.1-0ubuntu3~24.04.2
  - Tested on docker-compose version 1.29.2, build unknown
- Network access to VSP One Object endpoints
  - Bucket Access and Secret Keys have been created
  - Client credentials for Trino have been created
  - Table Bucket, Namespace, and the S3 Table have been created

Create the Directory Structure

```
mkdir -p trino-cluster
cd trino-cluster

# Create directory structure
mkdir -p {coordinator,worker1,worker2}/etc/catalog
mkdir -p data/{coordinator,worker1,worker2}
mkdir -p pem
```

Directory Structure

```
trino-cluster
├── coordinator
│   └── etc
│       ├── catalog
│       │   └── airlinedata.properties
│       ├── config.properties
│       ├── jvm.config
│       ├── log.properties
│       └── node.properties
├── data
│   ├── coordinator
│   ├── worker1
│   └── worker2
├── docker-compose.yml
├── Dockerfile
├── Dockerfile.superset
├── pem
│   ├── kc.pem
│   └── s3.pem
├── worker1
│   └── etc
│       ├── catalog
│       │   └── airlinedata.properties
│       ├── config.properties
│       ├── jvm.config
```

```
│       └── node.properties
└── worker2
    └── etc
        ├── catalog
        │   └── airlinedata.properties
        ├── config.properties
        ├── jvm.config
        └── node.properties
```

SSL Certificate Setup

In addition to the Bucket Access Key, Secret Access Key, and client credentials, the Keycloak and S3 CA certificates are required for Trino configuration. These Keycloak and S3 CA certificates will be added to the Trino container Java keystore. Adding the self-signed certs to Trino will prevent SSL verification errors.

To export the Keycloak CA certificate:

```
openssl s_client -connect admin.gms.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/kc.pem
```

To export the S3 CA certificate:

```
openssl s_client -connect s3.site02.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/s3.pem
```

After exporting the S3 and Keycloak certificates copy the `s3.pem` and `kc.pem` files into `~/trino-cluster/pem`.

Create the following coordinator configuration files and copy them into `~/trino-cluster/coordinator/etc`.

`config.properties`

```
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8080
discovery.uri=http://coordinator:8080
query.client.timeout=15m
```

`jvm.config`

```
-server
-Xmx2G
-XX:+UseG1GC
-XX:+ExplicitGCInvokesConcurrent
```

`log.properties`

```
io.trino=DEBUG
```

`node.properties` for worker1

```
node.environment=hitachi_labs
node.id=trino-worker-1
node.data-dir=/data
```

`node.properties` for worker2

```
node.environment=hitachi_labs
node.id=trino-worker-2
node.data-dir=/data
```

Create the Trino catalog properties file. Copy this file into:

`~/trino-cluster/coordinator/etc/catalog`, `~/trino-cluster/worker1/etc/catalog`, and `~/trino-cluster/worker2/etc/catalog`.

`airlinedata.properties`

```
 connector.name=iceberg
fs.native-s3.enabled=true
iceberg.catalog.type=rest
iceberg.file-format=PARQUET
iceberg.rest-catalog.uri=https://lakekeeper.s3.site02.100gvsp1o.company.com/catalog
iceberg.rest-catalog.oauth2.scope=s3catalog-scope
iceberg.rest-catalog.security=OAUTH2
iceberg.rest-catalog.oauth2.credential=trino-client:mXBr3PLyKBOJfTrhzUfkJw51kr3jCOpd
iceberg.rest-catalog.oauth2.server-uri=https://admin.gms.100gvsp1o.company.com/ui/
auth/realms/vsp-object/protocol/openid-connect/token
iceberg.rest-catalog.vended-credentials-enabled=false
iceberg.rest-catalog.warehouse=airlinedata
s3.aws-access-key=AKIA7X9B3vQWmTzKJrLp
s3.aws-secret-key=ZTk1NzYxMmEtYjYxMi00NjQyLTkzNzEtYzY2YjQw
s3.connection-max-idle-time=5m
s3.connection-ttl=1h
s3.endpoint=https://s3.site02.100gvsp1o.company.com
s3.path-style-access=true
s3.region=site02
s3.socket-connect-timeout=10s
s3.socket-read-timeout=10m
s3.tcp-keep-alive=true
```

Trino Dockerfile

```
# Use the official Trino image as the base image
FROM trinodb/trino:latest

# Copy the custom CA certificate into the container
COPY pem/s3new.pem /etc/trino/custom-ca/s3.pem
COPY pem/kc.pem /etc/trino/custom-ca/kc.pem
```

```
# Set the JAVA_HOME environment variable
ENV JAVA_HOME=/usr/lib/jvm/temurin/jdk-24.0.1+9

USER root

# Import the custom CA certificate into the Java keystore
RUN keytool -import -alias s3 -keystore $JAVA_HOME/lib/security/cacerts -file /etc/
trino/custom-ca/s3.pem -storepass changeit -noprompt
RUN keytool -import -alias kc -keystore $JAVA_HOME/lib/security/cacerts -file /etc/
trino/custom-ca/kc.pem -storepass changeit -noprompt
RUN keytool -import -alias s3 -keystore cacerts -file /etc/trino/custom-ca/s3.pem -
storepass changeit -noprompt
RUN keytool -import -alias kc -keystore cacerts -file /etc/trino/custom-ca/kc.pem -
storepass changeit -noprompt

USER trino

# Verify that the certificate was added
RUN keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit |
grep s3
RUN keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit |
grep kc
RUN keytool -list -keystore cacerts -storepass changeit | grep s3
RUN keytool -list -keystore cacerts -storepass changeit | grep kc
```

Superset `Dockerfile.superset`

```
FROM apache/superset:latest

# Switch to root to install packages
USER root

# Install Trino SQLAlchemy driver
RUN pip install trino

# Switch back to superset user
USER superset
```

Trino `docker-compose` file

```
version: "3.8"

services:
  coordinator:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: trino-coordinator
    ports:
      - "8080:8080"  # Trino web UI
```

```
    volumes:
      - ./coordinator/etc:/etc/trino
      - ./data/coordinator:/data
    environment:
      - DISCOVERY_PORT=8080
    restart: unless-stopped

  worker1:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: trino-worker1
    depends_on:
      - coordinator
    volumes:
      - ./worker1/etc:/etc/trino
      - ./data/worker1:/data
    restart: unless-stopped

  worker2:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: trino-worker2
    depends_on:
      - coordinator
    volumes:
      - ./worker2/etc:/etc/trino
      - ./data/worker2:/data
    restart: unless-stopped

  superset:
    build:
      context: .
      dockerfile: Dockerfile.superset
    container_name: superset
    ports:
      - "8088:8088"  # Superset web UI
    environment:
      - SUPERSET_SECRET_KEY=your-secret-key-here
    volumes:
      - superset_home:/app/superset_home
    depends_on:
      - coordinator
    restart: unless-stopped
    command: >
      sh -c "
      superset fab create-admin --username admin --firstname Superset --lastname
Admin –email admin@superset.com --password admin &&
      superset db upgrade &&
      superset init &&
```

```
      superset run -h 0.0.0.0 -p 8088 --with-threads --reload --debugger
      "
volumes:
  superset_home:
```

Build the Trino+Superset cluster by running the following Docker Compose commands from the `~/trino-cluster` directory:

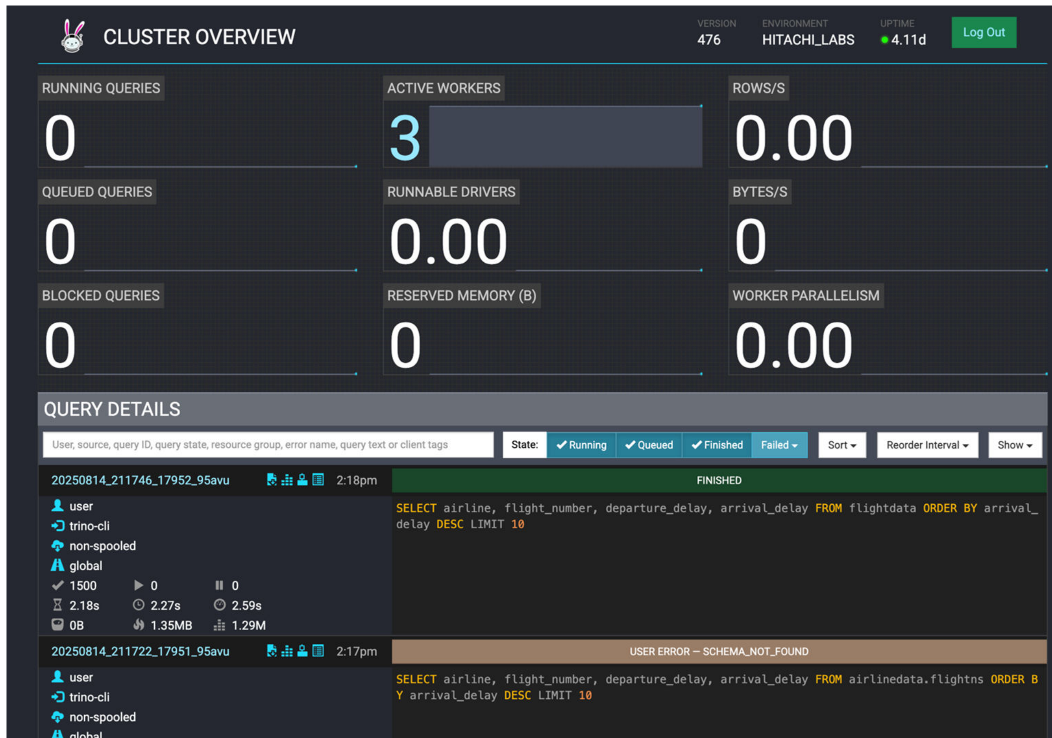
```
docker-compose build --no-cache
docker-compose up -d
```

Verify that the containers are running:

```
docker ps
user@User-S3-VM1:~/trino$ docker ps
CONTAINER ID   IMAGE                       COMMAND                  CREATED
STATUS               PORTS                                        NAMES
78e4394f0ea7   trino-cluster_superset      "sh -c ' superset fa…"   4 days ago   Up 4
days (healthy)   0.0.0.0:8088->8088/tcp, :::8088->8088/tcp   superset
c80390899d28   trino-cluster_worker2       "/usr/lib/trino/bin/…"   4 days ago   Up 4
days (healthy)   8080/tcp                                    trino-worker2
181f57fe0cf8   trino-cluster_worker1       "/usr/lib/trino/bin/…"   4 days ago   Up 4
days (healthy)   8080/tcp                                    trino-worker1
839ac14ea6de   trino-cluster_coordinator   "/usr/lib/trino/bin/…"   4 days ago   Up 4
days (healthy)   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   trino-coordinator
```

If there are no errors with the containers starting and running, go to `http://<trino-host-ip>:8080/ui/#` and log in (there is no login name; just enter an arbitrary name).

The Trino cluster overview should show three active workers:



Click the number “3” to view information about the active worker nodes.

WORKER LIST

VERSION 476 · ENVIRONMENT HITACHI_LABS · UPTIME 4.11d · Log Out

Overview

Node ID	Node IP	Node Version	Coordinator	State
trino-coordinator	172.18.0.2	476	true	active
trino-worker-1	172.18.0.4	476	false	active
trino-worker-2	172.18.0.5	476	false	active

Verify that the VSP One Object S3 Table can be queried through the Trino cluster:

```
import trino
import pandas as pd

host='172.23.91.22'
port=8080
user='admin'
catalog='airlinedata'
schema='flightns'
table = 'flightdata'

def trino_query(query, host, port, user,catalog, schema):
    conn = trino.dbapi.connect(host=host,port=port,user=user,catalog=catalog,
schema=schema)
    cur = conn.cursor()
    cur.execute(query)
    rows = cur.fetchall()
    columns = [desc[0] for desc in cur.description]
```

```
    df = pd.DataFrame(rows, columns=columns)
    cur.close()
    conn.close()
    return df

query = f"SELECT airline, flight_number, departure_delay, arrival_delay FROM {table}
ORDER BY arrival_delay DESC LIMIT 10"
df = trino_query(query, host, port, user,catalog, schema)
print(df)

             airline flight_number  departure_delay  arrival_delay
0    Delta Air Lines        DL1515              505            490
1    Alaska Airlines        AS4083              493            484
2  American Airlines         AA877              490            482
3    United Airlines        UA7447              488            480
4    United Airlines        UA8300              432            462
5    Spirit Airlines         NK784              437            458
6    United Airlines        UA6024              447            445
7    JetBlue Airways        B66159              435            443
8    JetBlue Airways        B69403              401            408
9    United Airlines        UA7011              390            401
```

Verify that data can be inserted using Trino:

```
query = f"""INSERT INTO {table} (
    flight_id, airline, flight_number, aircraft_type, departure_airport,
arrival_airport,
    scheduled_departure_time, scheduled_arrival_time, actual_departure_time,
actual_arrival_time,
    departure_delay, arrival_delay, passenger_count, ticket_price,
flight_duration_minutes,
    distance_miles, status, gate, pilot_id, crew_count, created_at, updated_at,
    data_source, record_version
) VALUES (
    'f123e456-789a-4bcd-ef01-234567890abc',
    'American Airlines',
    'AA1234',
    'Boeing 737-800',
    'JFK',
    'LAX',
    TIMESTAMP '2025-08-16 08:00:00',
    TIMESTAMP '2025-08-16 11:30:00',
    TIMESTAMP '2025-08-16 08:05:00',
    TIMESTAMP '2025-08-16 11:35:00',
    5,
    5,
    150,
    CAST(299.99 AS DECIMAL(8,2)),
    330,
```

```
    2475,
    'Completed',
    'A12',
    'p789-456a-123b-cdef-987654321abc',
    4,
    CURRENT_TIMESTAMP,
    CURRENT_TIMESTAMP,
    'manual_entry',
    1
)"""

df = trino_query(query, host, port, user, catalog, schema)
print(df)

   rows
0     1
```

Verify the new insert:

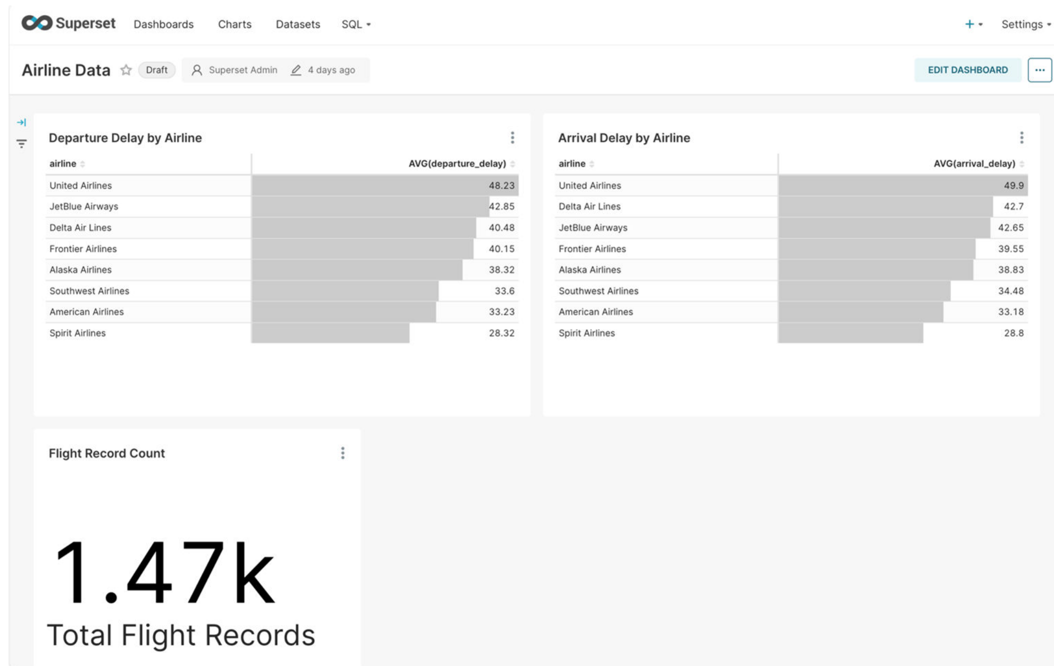
```
query = f"""SELECT airline, flight_number, scheduled_departure_time FROM {table}
WHERE flight_id = 'f123e456-789a-4bcd-ef01-234567890abc'"""
df = trino_query(query, host, port, user, catalog, schema)
print(df)

             airline flight_number scheduled_departure_time
0  American Airlines        AA1234      2025-08-16 08:00:00
```

Verify Superset with Trino and VSP One Object

1. Go to `http://<superset-host-IP>:8088/` and log in as `admin/admin`.

2. Within Superset navigate to Settings > Database Connections and click + Database.
    - From the Supported Databases dropdown, select Trino.
    - Specify a Display Name (for example: Airline Data).
    - Specify the SQLAlchemy URI: `trino://user@<superset-host-IP>:8080/airlinedata`.

After the Trino/VSP One Object S3 Table has been added to Superset, users can begin using data in the S3 Table to create charts and dashboards:



Appendix C: Detailed Spark Cluster Docker Deployment

The following describes how to build and deploy a Spark cluster with JupyterLab using Docker and Docker Compose for use with Hitachi VSP One Object store with S3 Table support.

Before installing and configuring Spark, make sure the following requirements have been met.

- Linux environment (tested on Ubuntu 24.04.2 LTS)
- Docker and Docker Compose installed
  - Tested on docker version 27.5.1, build 27.5.1-0ubuntu3~24.04.2
  - Tested on docker-compose version 1.29.2, build unknown
  - Network access to VSP One Object endpoints
  - Bucket Access and Secret Keys have been created
  - Client credentials for Spark have been created
  - Table Bucket, Namespace, and the S3 Table have been created

Create the Directory Structure

```
# Create directory structure
mkdir -p spark-cluster/pem
mkdir -p spark-cluster/work
```

Directory Structure

```
Spark-cluster
├── docker-compose.yml
├── Dockerfile
├── pem
|   ├── kc.pem
|   └── s3new.pem
├── requirements.txt
└── work
```

SSL Certificate Setup

In addition to the Bucket Access Key, Secret Access Key, and client credentials, the Keycloak and S3 CA certificates are required for the Spark configuration. These certificates must be imported into the Spark container Java keystore. Adding the self-signed certificates ensures that Spark can connect securely to VSP One Object endpoints without SSL verification errors.

Export the Keycloak CA certificate:

```
openssl s_client -connect admin.gms.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/kc.pem
```

Export the S3 CA certificate:

```
openssl s_client -connect s3.site02.100gvsp1o.company.com:443 < /dev/null 2>/dev/null
| openssl x509 > pem/s3.pem
```

After exporting the S3 and Keycloak certificates, copy the `s3.pem` and `kc.pem` files into `kc.pem files into ~/spark-cluster/pem`.

Spark Dockerfile

```
# ──────────────────────────────────────────────────────────────────────────────
# Spark 3.4.2 + Scala 2.12 + Java 17 + Python 3.11
# ──────────────────────────────────────────────────────────────────────────────

ARG BASE_IMAGE=quay.io/okdp/spark-py:spark-3.4.2-python-3.11-scala-2.12-java-17-2025-
05-20-1.0.1
FROM ${BASE_IMAGE}

USER root

# System packages (git for jupyterlab-git, curl for downloads)
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
      sudo \
      git \
      curl \
      unzip \
```

```
      build-essential \
      software-properties-common && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

# Python deps
COPY requirements.txt /tmp/requirements.txt
RUN pip install --no-cache-dir -r /tmp/requirements.txt

# Optional: Scala kernel for notebooks (leave if you use it)
RUN python3 -m spylon_kernel install || true

# ──────────────────────────────────────────────────────────────────────────────
# Iceberg 1.6.1 runtime for Spark 3.4 + Scala 2.12
# ──────────────────────────────────────────────────────────────────────────────
ENV SPARK_HOME=/opt/spark
RUN mkdir -p ${SPARK_HOME}/jars
# Runtime (includes Spark SQL/DataSource V2 adapter)
RUN curl -L -o ${SPARK_HOME}/jars/iceberg-spark-runtime-3.4_2.12-1.6.1.jar \
  https://repo1.maven.org/maven2/org/apache/iceberg/iceberg-spark-runtime-
3.4_2.12/1.6.1/iceberg-spark-runtime-3.4_2.12-1.6.1.jar
# AWS bundle (S3FileIO + AWS SDKs)
RUN curl -L -o ${SPARK_HOME}/jars/iceberg-aws-bundle-1.6.1.jar \
  https://repo1.maven.org/maven2/org/apache/iceberg/iceberg-aws-bundle/1.6.1/iceberg-
aws-bundle-1.6.1.jar

# Optional: AWS CLI
RUN curl -sSL "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip" \
 && unzip -q awscliv2.zip \
 && ./aws/install \
 && rm -rf aws awscliv2.zip

# spark-defaults.conf: pre-load both Iceberg jars
RUN mkdir -p ${SPARK_HOME}/conf && \
    echo "spark.jars ${SPARK_HOME}/jars/iceberg-spark-runtime-3.4_2.12-1.6.1.jar,$
{SPARK_HOME}/jars/iceberg-aws-bundle-1.6.1.jar" > ${SPARK_HOME}/conf/spark-
defaults.conf && \
    echo "# You can also set stable cluster-wide defaults here later (catalog, S3,
etc.)" >> ${SPARK_HOME}/conf/spark-defaults.conf

# ──────────────────────────────────────────────────────────────────────────────
# Custom CA certificates (both into system JDK cacerts and a local file "cacerts")
# ──────────────────────────────────────────────────────────────────────────────
RUN mkdir -p /etc/spark/custom-ca
COPY pem/s3new.pem /etc/spark/custom-ca/s3.pem
COPY pem/kc.pem    /etc/spark/custom-ca/kc.pem

# Import the custom CA certificate into the Java keystore
RUN keytool -import -alias s3 -keystore $JAVA_HOME/lib/security/cacerts -file /etc/
spark/custom-ca/s3.pem -storepass changeit -noprompt
RUN keytool -import -alias kc -keystore $JAVA_HOME/lib/security/cacerts -file /etc/
```

```
spark/custom-ca/kc.pem -storepass changeit -noprompt
RUN keytool -import -alias s3 -keystore cacerts -file /etc/spark/custom-ca/s3.pem -
storepass changeit -noprompt
RUN keytool -import -alias kc -keystore cacerts -file /etc/spark/custom-ca/kc.pem -
storepass changeit -noprompt

# Verify that the certificate was added
RUN keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit |
grep s3
RUN keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit |
grep kc
RUN keytool -list -keystore cacerts -storepass changeit | grep s3
RUN keytool -list -keystore cacerts -storepass changeit | grep kc

# Workspace
RUN mkdir -p /work
WORKDIR /work

# Ports
# 9090 = master UI (remapped from 8080), 9091 = worker UI (remapped from 8081)
# 7077 = master RPC, 8888 = Jupyter
EXPOSE 9090 9091 7077 8888

CMD ["/bin/bash"]
```

Spark docker-compose file:

```
services:
  spark-master:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: spark-master
    hostname: spark-master
    command: ["/opt/spark/bin/spark-class", "org.apache.spark.deploy.master.Master",
              "--host", "spark-master", "--port", "7077", "--webui-port", "8080"]
    environment:
      - SPARK_NO_DAEMONIZE=true
      - SPARK_LOG_DIR=/tmp
      - PYSPARK_PYTHON=python3
      - PYSPARK_DRIVER_PYTHON=python3
    ports:
      - "7077:7077"   # Spark master RPC
      - "9090:8080"   # Master UI
    networks:
      - sparknet
    volumes:
      - ./work:/work

  spark-worker:
```

```
    build:
      context: .
      dockerfile: Dockerfile
    container_name: spark-worker
    hostname: spark-worker
    depends_on:
      - spark-master
    command: ["/opt/spark/bin/spark-class", "org.apache.spark.deploy.worker.Worker",
              "spark://spark-master:7077",
              "--webui-port", "8081"]
    environment:
      - SPARK_NO_DAEMONIZE=true
      - SPARK_LOG_DIR=/tmp
      - SPARK_WORKER_CORES=8           # tweak as needed
      - SPARK_WORKER_MEMORY=16G         # tweak as needed
      - PYSPARK_PYTHON=python3
      - PYSPARK_DRIVER_PYTHON=python3
    ports:
      - "9091:8081"   # Worker UI
    networks:
      - sparknet
    volumes:
      - ./work:/work

  jupyter:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: spark-jupyter
    hostname: spark-jupyter
    depends_on:
      - spark-master
    environment:
      # point PySpark to the cluster master
      - SPARK_MASTER=spark://spark-master:7077
      - SPARK_HOME=/opt/spark
      - PYSPARK_PYTHON=python3
      - PYSPARK_DRIVER_PYTHON=jupyter
      - PYSPARK_DRIVER_PYTHON_OPTS=lab --ip=0.0.0.0 --no-browser --NotebookApp.token=
--NotebookApp.password=
      # Keep Spark logs simple in notebooks
      - SPARK_LOG_DIR=/tmp
      # optional: avoid driver UI port conflicts when you run multiple jobs
      - SPARK_UI_PORT=4040
    command: ["/bin/bash", "-lc", "jupyter lab --allow-root --NotebookApp.token='' --
NotebookApp.password='' --ip=0.0.0.0 --no-browser --NotebookApp.allow_origin='*' --
NotebookApp.disable_check_xsrf=True --port=8888"]
    ports:
      - "8888:8888"   # JupyterLab
    networks:
      - sparknet
```

```
    volumes:
      - ./work:/work

networks:
  sparknet:
    driver: bridge
```

requirements.txt file:

```
ipywidgets==8.*
findspark==2.*
jupyter==1.1.1
jupyterlab-git==0.*
jupyter-console==6.6.3
jupyter-events==0.12.0
jupyter-lsp==2.2.6
jupyter_client==8.6.3
jupyter_core==5.8.1
jupyter_server==2.16.0
jupyter_server_terminals==0.5.3
jupyterlab==4.4.5
jupyterlab_pygments==0.3.0
jupyterlab_server==2.27.3
jupyterlab_widgets==3.0.15
faker
numpy
pandas
pyarrow
scipy
scikit-learn
spylon-kernel
pyiceberg
pyiceberg[pyarrow,duckdb,pandas]
jupysql
matplotlib
grpcio
pycolor
papermill
```

Build the Spark+JupyterLab cluster, run the following docker-compose commands from the `~/spark-cluster` directory:

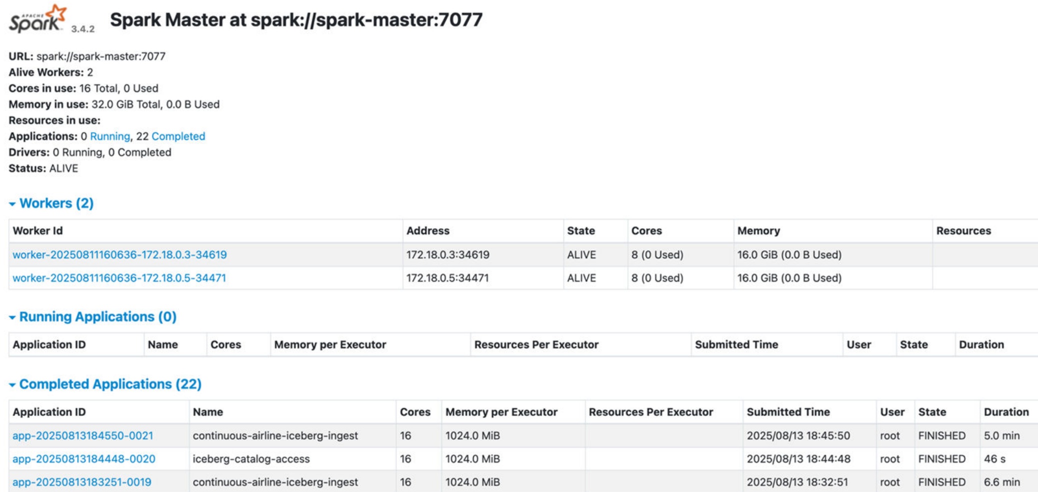
```
docker-compose build --no-cache
docker-compose up -d
```

Verify that the containers are running:

```
docker ps
user@user-S3-VM3:~$ docker ps
CONTAINER ID   IMAGE                 COMMAND                    CREATED
```

```
STATUS
PORTS
                                        NAMES
8cb1e8e2d8ca   spark_jupyter        "/opt/entrypoint.sh …"   4 days ago   Up 4
days   7077/tcp, 9090-9091/tcp, 0.0.0.0:8888->8888/tcp, :::8888-
>8888/tcp                                                   spark-jupyter
fd3351ab6772   spark_spark-worker1   "/opt/entrypoint.sh …"   4 days ago   Up 4
days   7077/tcp, 8888/tcp, 9090-9091/tcp, 0.0.0.0:9091->8081/tcp, [::]:9091-
>8081/tcp                                       spark-worker1
2f24ce4cf51a   spark_spark-worker2   "/opt/entrypoint.sh …"   4 days ago   Up 4
days   7077/tcp, 8888/tcp, 9090-9091/tcp, 0.0.0.0:9092->8081/tcp, [::]:9092-
>8081/tcp                                       spark-worker2
66c4077fcf2a   spark_spark-master    "/opt/entrypoint.sh …"   4 days ago   Up 4
days   8888/tcp, 0.0.0.0:7077->7077/tcp, :::7077->7077/tcp, 9090-9091/tcp,
0.0.0.0:9090->8080/tcp, [::]:9090->8080/tcp   spark-master
```

If there are no errors with the containers starting and running, go to `http://<spark-host-IP>:9090/` to verify the spark-master node:



Verify that the VSP One Object S3 Table can be queried from the Spark cluster.

To begin, from a notebook or a python script setup the spark session:

```
import os
import findspark
findspark.init("/opt/spark")

import random
import pyspark
from pyspark.sql import SparkSession, Row
from pyspark.sql.types import (StructType, StructField, StringType,
                                IntegerType, DateType, DecimalType,
                                DoubleType, TimestampType, BooleanType)
from decimal import Decimal
from datetime import datetime, date
```

```
os.environ['JAVA_TOOL_OPTIONS'] = '-Djavax.net.ssl.trustStore=/work/cacerts -
Djavax.net.ssl.trustStorePassword=changeit'

catalog_name = "airlinedata"
namespace_name = "flightns"
table_name = "flightdata"
region = "site02"
s3_endpoint = "https://s3.site02.100gvsp1o.company.com"
lakekeeper_url = "https://lakekeeper.s3.site02.100gvsp1o.company.com/catalog"
token_url = "https://admin.gms.100gvsp1o.company.com/ui/auth/realms/vsp-object/
protocol/openid-connect/token"
oauth_credential = "spark-clien:mXBr3PLyKBOJfTrhzUfkJw51kr3jCOpd"

# Get Spark version for JAR compatibility
spark_version = pyspark.__version__
spark_min_version = ".".join(spark_version.split(".")[:2])  # e.g., "3.4.2"
iceberg_version = "1.6.1"

# Build Spark session with Iceberg configuration
spark = (
    SparkSession.builder
    .appName("iceberg-catalog-access")
    .master("spark://spark-master:7077") # .master("local[*]")
    .config("spark.pyspark.python", "python3")
    .config("spark.pyspark.driver.python", "python3")

    # Enable Iceberg extensions
    .config("spark.sql.extensions",
"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions")

    # Catalog configuration
    .config(f"spark.sql.catalog.{catalog_name}",
"org.apache.iceberg.spark.SparkCatalog")
    .config(f"spark.sql.catalog.{catalog_name}.uri", lakekeeper_url)
    .config(f"spark.sql.catalog.{catalog_name}.warehouse", catalog_name)
    .config(f"spark.sql.catalog.{catalog_name}.type", "rest")
    .config(f"spark.sql.catalog.{catalog_name}.scope", "s3catalog-scope")

    # Authentication settings
    .config(f"spark.sql.catalog.{catalog_name}.credential", f"{data['client_id']}:
{data['client_secret']}")
    .config(f"spark.sql.catalog.{catalog_name}.grant-type", data["grant_type"])
    .config(f"spark.sql.catalog.{catalog_name}.oauth2-server-uri", token_url)
    .config(f"spark.sql.catalog.{catalog_name}.header.X-Iceberg-Access-Delegation",
"vendedcredentials")

    # SSL trust store configuration
    .config("spark.driver.extraJavaOptions", "-Djavax.net.ssl.trustStore=cacerts -
Djavax.net.ssl.trustStorePassword=changeit")
    .config("spark.executor.extraJavaOptions", "-Djavax.net.ssl.trustStore=cacerts -
Djavax.net.ssl.trustStorePassword=changeit")
```

```
    .getOrCreate()
)
```

Verify that the S3 Table can be queried:

```
query = f"""SELECT
                airline, flight_number, departure_delay, arrival_delay
            FROM {namespace_name}.{table_name}
            ORDER BY
                arrival_delay
            DESC LIMIT 10"""

spark.sql(query).show()

+-----------------+-------------+---------------+-------------+
|          airline|flight_number|departure_delay|arrival_delay|
+-----------------+-------------+---------------+-------------+
|  Delta Air Lines|       DL1515|            505|          490|
|  Alaska Airlines|       AS4083|            493|          484|
|American Airlines|        AA877|            490|          482|
|  United Airlines|       UA7447|            488|          480|
|  United Airlines|       UA8300|            432|          462|
|  Spirit Airlines|        NK784|            437|          458|
|  United Airlines|       UA6024|            447|          445|
|  JetBlue Airways|       B66159|            435|          443|
|  JetBlue Airways|       B69403|            401|          408|
|  United Airlines|       UA7011|            390|          401|
+-----------------+-------------+---------------+-------------+
```

Verify that Spark can insert into the table:

```
insert_query = f"""
INSERT INTO {catalog_name}.{namespace_name}.flightdata VALUES (
    'FL001',
    'American Airlines',
    'AA1234',
    'Boeing 737',
    'LAX',
    'JFK',
    TIMESTAMP '2025-08-18 08:00:00',
    TIMESTAMP '2025-08-18 16:30:00',
    TIMESTAMP '2025-08-18 08:15:00',
    TIMESTAMP '2025-08-18 16:45:00',
    15,
    15,
    180,
    450.00,
    330,
    2475,
```

```
    'Completed',
    'A12',
    'P001',
    6,
    TIMESTAMP '2025-08-18 09:00:00',
    TIMESTAMP '2025-08-18 09:00:00',
    'manual_entry',
    1
)
"""

spark.sql(insert_query)
```

Verify that the new record has been inserted:

```
query = f"""SELECT
            flight_id, airline, flight_number
          FROM {namespace_name}.{table_name}
          WHERE flight_number = 'AA1234' AND  flight_id = 'FL001'"""

spark.sql(query).show()

+---------+-----------------+-------------+
|flight_id|          airline|flight_number|
+---------+-----------------+-------------+
|    FL001|American Airlines|       AA1234|
+---------+-----------------+-------------+
```

Appendix D: AWS CLI Examples

The following AWS CLI examples illustrate how to manage S3 Tables in Hitachi VSP One Object.

Create S3 Table Bucket

```
aws s3tables create-table-bucket --name tests3tablebucket --region site02 --no-verify-
ssl --endpoint https://s3.site02.100gvsp1o.company.com
```

Get Table Bucket List

```
aws s3tables list-table-buckets --endpoint https://s3.site02.100gvsp1o.company.com --
no-verify-ssl
```

Create Namespace

```
aws s3tables create-namespace --table-bucket-arn arn:aws:s3tables:site02:-
1755417697:bucket/tests3tablebucket --namespace ns1 --region site02 --no-verify-ssl --
endpoint https://s3.site02.100gvsp1o.company.com
```

Get Namespace

```
aws s3tables get-namespace --table-bucket-arn arn:aws:s3tables:site02:-
1755417697:bucket/tests3tablebucket --namespace ns1 --region site02 --no-verify-ssl --
endpoint https://s3.site02.100gvsp1o.company.com
```

Create Table

```
aws s3tables create-table --cli-input-json '{
    "tableBucketARN": "arn:aws:s3tables:site02:-1755417697:bucket/tests3tablebucket",
    "namespace": "ns1",
    "name": "employee3",
    "format": "ICEBERG",
    "metadata": {
        "iceberg": {
            "schema": {
                "fields": [
                    {
                        "name": "name",
                        "type": "string",
                        "required": true
                    },
                    {
                        "name": "age",
                        "type": "int8",
                        "required": false
                    },
                    {
                        "name": "mob",
                        "type": "string",
                        "required": false
                    }
                ]
            }
        }
    }
}'  --region site02 --no-verify-ssl --endpoint https://s3.site02.100gvsp1o.company.com
```

Get Table

```
aws s3tables get-table --table-bucket-arn arn:aws:s3tables:site02:-1755417697:bucket/
tests3tablebucket --namespace ns1 --name employee3 --region site02 --no-verify-ssl --
endpoint https://s3.site02.100gvsp1o.company.com
```

Appendix E: Python Examples

The following is a python examples demonstrating how to manage Hitachi VSP One Object S3 Tables.

These examples, use the AWS python boto3 library. For more information on how to use the python boto3 library, visit the boto3 documentation page: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

Define the endpoints, access and secret keys, and declare the boto3 python client:

```
endpoint_url = 'https://s3.site02.100gvsp1o.company.com'
access_key_id = <your-access-key>
secret_access_key = <your-secret-key>
region_name = 'site02'
# verify = False
verify = 's3.pem'
table_bucket = 'tests3tablebucket'
namespace = 'ns1'
table = 'employee3'
table_name = "flightsdata"


client = boto3.client(
    's3tables',
    endpoint_url=endpoint_url,
    aws_access_key_id=access_key_id,
    aws_secret_access_key=secret_access_key,
    region_name=region_name,
    verify=verify)
```

Create S3 Table Bucket

```
resp = client.create_table_bucket(name=table_bucket) # table_bucket defined above
ARN = resp['arn'] # Capture the bucket ARN
```

Get Bucket List

```
response = client.list_table_buckets()

for bucket in response.get("tableBuckets", []):
    print(f"Remaining bucket: {bucket['name']} → ARN: {bucket['arn']}")
```

Create Namespace

```
s3.create_namespace(
    tableBucketARN=ARN, # ARN defined in initial bucket creation above
    namespace=[namespace] # namespace defined above
)
```

List Namespaces

```
response = client.list_namespaces(tableBucketARN=ARN) # ARN defined in initial bucket
creation above
```

```
for ns in response.get("namespaces", []):
    print(f"{ns['namespace'][0]} → ID: {ns['namespaceId']} created at
{ns['createdAt']}")
```

Create Table

```
schema =  {
        "iceberg": {
            "schema": {
                "fields": [
                    {"name": "ID", "type": "int8", "required": True},
                    {"name": "name", "type": "string", "required": False},
                    {"name": "age", "type": "int8", "required": False},
                    {"name": "salary", "type": "double", "required": False}
                ]
            }
        }
    }

resp = client.create_table(
    tableBucketARN=ARN, # ARN defined in initial bucket creation above
    namespace=namespace, # namespace defined above
    name="employee3",
    format="ICEBERG",
    metadata=schema
)
tableARN = resp['tableARN'] # capture the table ARN
```

Get Table

```
response = client.get_table(
    tableBucketARN=ARN,
    namespace=namespace,
    name=table_name
)

for k,v in response.items():
    print(f"{k}: {v}")
```

List Tables

```
response = client.list_tables(
    tableBucketARN=ARN,      # ARN defined in initial bucket creation above
    namespace=namespace,     # namespace defined above
)

for tbl in response.get("tables", []):
    print(f"{tbl['namespace'][0]}.{tbl['name']} (ARN: {tbl['tableARN']}) created at
{tbl['createdAt']}")
```

Appendix F: Troubleshooting and Tips

Trino troubleshooting

Common issues:

SSL Certificate Errors:

- Symptom: SSL handshake failures in logs
- Solution: Verify certificates are properly imported
- Check: `docker exec -it trino-coordinator keytool -list -keystore $JAVA_HOME/lib/security/cacerts | grep -E "(s3|kc)"`

Catalog Not Found Errors:

- Symptom: No catalog errors or Trino does not display the catalog

  Solution: Ensure catalog properties exist on all nodes

  Check:

  - Verify that the catalog properties file is copied to the coordinator and all worker nodes
    - `ls -la coordinator/etc/catalog/ worker1/etc/catalog/ worker2/etc/catalog/`
    - `docker exec -it trino-coordinator ls /etc/trino/catalog/`
      - Verify on both coordinator and worker nodes
  - Verify the catalog properties file is correct
    - Verify that the endpoints are correct and the access and secret keys are correct

OAuth2 Authentication Failures

- Symptom: 401/403 errors in logs
- Solution: Verify OAuth2 credentials and endpoints
- Check: Verify the endpoints and client credentials are correct

Connection timeouts

- Symptom: Timeout errors connecting to S3 endpoints
- Solution: Check network connectivity, firewall rules and DNS
- Check: `docker exec -it trino-coordinator ping s3.site02.100gvsp1o.company.com`

Log analysis

```
# View real-time logs
docker-compose logs -f coordinator # check workers too

# Search for specific errors
docker-compose logs coordinator | grep -i error  # check workers too
docker-compose logs coordinator | grep -i ssl    # check workers too
docker-compose logs coordinator | grep -i oauth  # check workers too
```

Health checks

```
-- Test catalog connectivity
SHOW CATALOGS;

-- Test schema access
SHOW SCHEMAS FROM airlinedata;

-- Test table operations
DESCRIBE airlinedata.flightns.flightdata;

-- Test distributed operations
SELECT flight_id, COUNT(*) as flight_count
FROM airlinedata.flightns.flightdata
GROUP BY department;
```

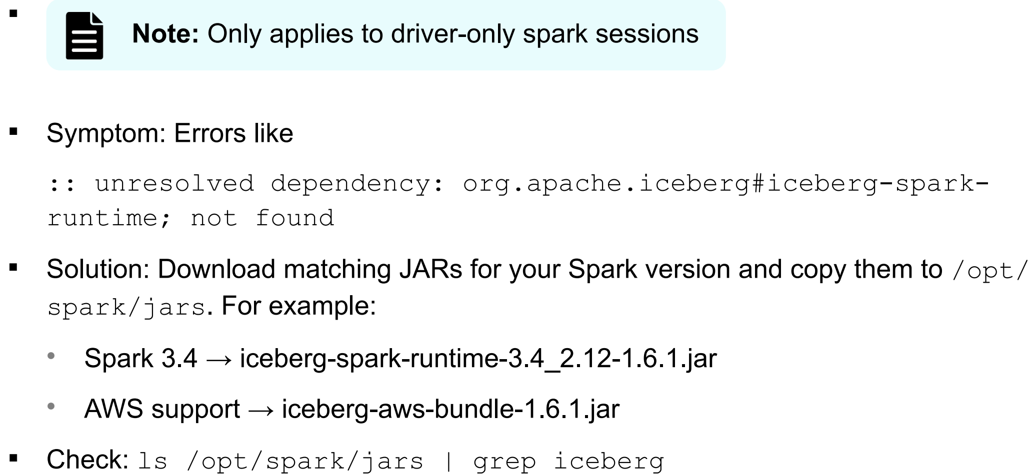
Spark Troubleshooting

Common issues

SSL certificate errors

- Symptom: SSL handshake failures when Spark connects to S3 endpoints or the REST catalog (LakeKeeper/Keycloak).
- Solution: Import the Hitachi S3 and Keycloak CA certificates into the Java trust store inside all Spark containers/VMs ($JAVA_HOME/lib/security/cacerts).
- Check: `docker exec -it spark-master keytool -list -keystore $JAVA_HOME/lib/security/cacerts -storepass changeit | grep -E "(s3|kc)"`

Missing Iceberg JARs

- 

  Note: Only applies to driver-only spark sessions

- Symptom: Errors like

  `:: unresolved dependency: org.apache.iceberg#iceberg-spark-runtime; not found`

- Solution: Download matching JARs for your Spark version and copy them to `/opt/spark/jars`. For example:
  - Spark 3.4 → iceberg-spark-runtime-3.4_2.12-1.6.1.jar
  - AWS support → iceberg-aws-bundle-1.6.1.jar
- Check: `ls /opt/spark/jars | grep iceberg`

Catalog not visible

- Symptom: Running `SHOW CATALOGS` in Spark only shows `spark_catalog`. Your REST catalog (for example, airlinedata) does not appear.

  Solution: This is expected with Iceberg REST catalogs in Spark. The catalog is usable even if not listed. Confirm by showing namespaces or tables.

  Check: `spark.sql("SHOW NAMESPACES IN airlinedata").show()`

Executors missing configurations

- Symptom: Queries succeed on the driver but fail on workers with S3 or REST catalog errors.
- Solution: Broadcast configs to executors by setting them in:
  - spark-defaults.conf inside the image
- Check: `docker exec -it spark-worker printenv | grep JAVA_TOOL_OPTIONS`

Wrong Spark/Iceberg Version Combo

- Symptom: `TypeError: 'JavaPackage' object is not callable or Spark fails to load Iceberg extensions.`
- Solution: Use version-matched components:
  - Spark 3.4.x + Iceberg 1.6.x
  - Spark 3.5.x + Iceberg 1.9.x
- Check: `spark.version # confirm Spark version`

Log analysis

```
# View real-time logs
docker-compose logs -f spark-master # check workers too

# Search for specific errors
docker-compose logs spark-master | grep -i error  # check workers too
```

```
docker-compose logs spark-master | grep -i ssl    # check workers too
docker-compose logs spark-master | grep -i oauth  # check workers too
```

Hitachi Vantara

Corporate Headquarters
2535 Augustine Drive
Santa Clara, CA 95054 USA

HitachiVantara.com/contact