Data Lake Insight

Service Bulletin

Issue	01
Date	2025-02-27

HUAWEI TECHNOLOGIES CO., LTD.

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 Product Bulletin

1.1 EOS Announcement for DLI Spark 3.1.1

Description

Huawei Cloud initiated an end of service (EOS) for DLI Spark 3.1.1 at 00:00:00 (GMT+08:00) on December 31, 2024.

Impact

After the EOS, no technical support will be provided for DLI Spark 3.1.1. You are advised to select the Spark engine of the latest version when executing jobs. DLI Spark 3.3.1 is recommended.

For jobs that are using DLI Spark 3.1.1, switch to the Spark engine of the latest version as soon as possible. Otherwise, no technical support will be provided if an error occurs during job execution.

If you have any questions or suggestions, please **submit a service ticket** or call us on +86-4000-955-988 or +86-950-808.

FAQ

- **How does the EOS affect the jobs that are using DLI Spark 3.1.1?**

  If a queue is created after the EOS of Spark 3.1.1, the compute engine that has reached EOS cannot be selected during job execution.

  Historical queues can still use Spark 2.3.2 to execute jobs. However, if an error occurs during job execution, no technical support is provided. Replace the compute engine with a new version as soon as possible.
- **Which version can be used as a replacement after the EOS?**

  DLI Spark 3.3.1 is recommended.
- **What are the advantages of DLI Spark 3.3.1?**

Table 1-1 Advantages of Spark 3.3.1

Feature	Description
Native performance acceleration	Improved the performance of Spark query statements.
Metadata access performance improvement	Improved Spark's metadata access performance for handling big data and enhanced data processing efficiency.
Improving the performance of OBS Committer when writing small files	Improved the performance of Object Storage Service (OBS) when writing small files, improving data transfer efficiency.
Dynamic executor shuffle data optimization	Improved the stability of resource scaling and cleaned up Executors when shuffle files are no longer needed.
Merging small files	If a large number of small files are generated during SQL execution, job execution and table query will take a long time. In this case, you are advised to merge small files. Merge small files by referring to **How Do I Merge Small Files?**
Modifying column comments of non-partitioned or partitioned tables	You can modify the column comments of non-partitioned or partitioned tables.
Collecting statistics on the CPU usage of SQL jobs	You can view the total CPU used on the console.
Viewing Spark logs of container clusters	You need to view logs in the container.
Dynamic UDF loading (OBT)	The UDF takes effect without restarting the queue.
Supporting flame graphs on the Spark UI	Flame graphs can be created on the Spark UI.
Optimizing the query performance of the **NOT IN** statement for SQL jobs	The query performance of the **NOT IN** statement is improved.
Optimizing the query performance of the **Multi-INSERT** statement	The query performance of the **Multi-INSERT** statement is improved.

- **Does the upgrade affect the DLI resource price?**

DLI bills you based on the amount of compute and storage resources consumed by jobs, regardless of the compute engine version.

- **How do I upgrade DLI Spark to version 3.1.1?**
  a. On the DLI management console, buy an elastic resource pool and create queues within the pool to provide compute resources required for job execution.
  b. In the navigation pane on the left, choose **Job Management** > **Spark Jobs**. On the displayed page, click **Create Job** in the upper right corner.
  c. On the displayed page, locate the target job and click **Edit** in the **Operation** column.
  d. On the page displayed, select the latest Spark version. Spark 3.3.1 is recommended.

1.2 EOL Announcement for DLI Yearly/Monthly and Pay-per-Use Queues as Well as Queue CUH Packages

Description

To improve resource sharing and increase the utilization of compute resources, the DLI team is upgrading yearly/monthly and pay-per-use queues to elastic resource pool queues. This means if you need to use DLI compute resources, you will need to buy an elastic resource pool and create queues within it.

- Huawei Cloud initiated an end of marketing (EOM) for DLI queues billed in yearly/monthly and pay-per-use modes and queue CUH packages on March 31, 2024, at 00:00:00 (GMT+08:00).
- Huawei Cloud schedules an end of life (EOL) for DLI queues billed in yearly/monthly and pay-per-use modes and queue CUH packages on June 30, 2025, at 00:00:00 (GMT+08:00).

Impact

- Once the EOM is reached, new DLI queues billed in yearly/monthly and pay-per-use modes and queue CUH packages, cannot be purchased.

  Until June 30, 2024 at 00:00:00 (GMT+08:00), you can renew your queues for up to one year or modify their specifications to meet your service needs.

  After June 30, 2024, at 00:00:00 (GMT+08:00), it will no longer be possible to renew or change your queues.
- Once the EOL is reached, queues will no longer be usable. Therefore, it is necessary to use an elastic resource pool or the **default** queue before the EOL. We recommend purchasing an elastic resource pool and creating queues within it to enjoy a wider range of DLI product capabilities.

If you have any questions or suggestions, please **submit a service ticket** or call us on +86-4000-955-988 or +86-950-808.

Lifecycle of DLI Yearly/Monthly and Pay-per-Use Queues

Table 1-2 Lifecycle of DLI yearly/monthly and pay-per-use queues

Function	Status	EOM Date	EOL Date
DLI yearly/ monthly and pay-per-use queues	EOM	March 31, 2024	June 30, 2025

NOTE

- EOM: indicates that the sales of this function are stopped.
- EOL: indicates that all sales and service activities are stopped.

FAQ

- **What will happen to jobs running on DLI queues with yearly/monthly or pay-per-use billing after the EOM?**

  Once the EOM is reached, new queues cannot be purchased.

  – Short-term solution: Until June 30, 2024 at 00:00:00 (GMT+08:00), you can renew your queues for up to one year or modify their specifications to meet your service needs.

  – Long-term solution: Use the elastic resource pool or **default** queue. You are advised to purchase an elastic resource pool and create queues in the pool.

- **What will happen to jobs running on DLI queues with yearly/monthly or pay-per-use billing after the EOL?**

  Once the EOL is reached, queues billed in yearly/monthly or pay-per-use mode will no longer be able to execute jobs.

  Before the EOL, you need to move your jobs from yearly/monthly and pay-per-use queues to queues in an elastic resource pool. To do this, you will need to purchase an elastic resource pool, create a queue within the pool, and then run your jobs on that queue.

- **What function can be used as a replacement after the EOM and EOL?**

  If you are running jobs on queues billed in yearly/monthly or pay-per-use mode, use an elastic resource pool or the **default** queue as soon as possible.

  – For pay-per-use dedicated queues, they can be directly moved to newly purchased elastic resource pools.

  – For yearly/monthly or pay-per-use non-dedicated queues, you will need to unsubscribe from them first, purchase new elastic resource pools, and then execute jobs in those pools.

- **What are the advantages of elastic resource pool queues compared with yearly/monthly and pay-per-use queues?**

  – Yearly/Monthly and pay-per-use queues: Such queues have predetermined resource specifications. However, if a job's resource

  requirements fluctuate, the queue resources may either go to waste or prove to be insufficient.
  - Elastic resource pool queue: Dynamic scaling improves resource utilization.

- **Do I have to pay for an elastic resource pool? Is a queue created in an elastic resource pool charged separately?**

  Elastic resource pools support the pay-per-use and package billing modes. For more information about the billing, see *Data Lake Insight Billing*.

  Queues added to an elastic resource pool are not billed separately, but be included in the billing for the elastic resource pool.
  - Pay-per-use: You are billed based on the actual CUs of the elastic resource pool.
  - Yearly/Monthly: You are billed based on the actual CUs of the elastic resource pool, with the specification part billed yearly/monthly and any excess billed on a pay-per-use basis.
  - Elastic resource pool CUH package: You are billed based on the price of the purchased package, with the specification within the package billed on a yearly/monthly basis. Any excess beyond the package specification will be billed on a pay-per-use basis.

  For more billing information about elastic resource pools, see **Billing for Elastic Resource Pools**.

- **How do I create an elastic resource pool queue?**
  a. **Buy an elastic resource pool.**
     i. On the DLI management console, choose **Resources** > **Resource Pool**.
     ii. On the **Resource Pool** page, click **Buy Resource Pool** in the upper right corner.

        Set parameters as instructed, click **Buy**, confirm the configuration, and click **Submit**.
  b. **Create a queue in the elastic resource pool.**

     Create one or more queues in the elastic resource pool to run jobs.
     i. Switch to the **Resource Pool** page.
     ii. Locate the target elastic resource pool and click **Add Queue** in the **Operation** column.
     iii. On the **Add Queue** page, configure basic queue information. Click **OK**.
  c. **Create a job.**

     Create a job and run it on the queue you have created.

Announcement published on: November 21, 2023

1.3 EOS Announcement for DLI Flink 1.10 and Flink 1.11

Description

Huawei Cloud initiated an end of service (EOS) for DLI Flink 1.10 and Flink 1.11 at 00:00:00 (GMT+08:00) on December 31, 2023.

Impact

After the EOS, no technical support will be provided for DLI Flink 1.10 and Flink 1.11. You are advised to select the Flink engine of the latest version when executing jobs. DLI Flink 1.15 is recommended.

For jobs that are using Flink 1.10 or Flink 1.11, switch to the Flink engine of the latest version as soon as possible. Otherwise, no technical support will be provided if an error occurs during job execution.

If you have any questions or suggestions, please **submit a service ticket** or call us on +86-4000-955-988 or +86-950-808.

FAQ

- **How does the EOS affect the jobs that are using Flink 1.10 or 1.11?**

  If a queue is created after the EOS of Flink 1.10 or 1.11, the compute engine that has reached EOS cannot be selected during job execution.

  Historical queues can still use Flink 1.10 or Flink 1.11 to execute jobs. However, if an error occurs during job execution, no technical support is provided. Replace the compute engine with a new version as soon as possible.
- **Which version can be used as a replacement after the EOS?**

  DLI Flink 1.15 is recommended.
- **What are the advantages of Flink 1.15?**
  - The syntax design of Flink 1.15 has been improved to achieve higher compatibility and consistency with mainstream open-source technology standards.
  - Flink 1.15 has added support for new connectors such as Hive and Hudi.

  For more advantages, see **Flink 1.15 Upgrade Guide**.
- **Does the upgrade of Flink affect the DLI resource price?**

  DLI bills you based on the amount of compute and storage resources consumed by jobs, regardless of the compute engine version.
- **How do I upgrade Flink to version 1.15?**
  a. On the DLI management console, buy an elastic resource pool and create queues within the pool to provide compute resources required for job execution.
  b. Log in to the DLI management console. In the navigation pane on the left, choose **Job Management** > **Flink Jobs**.

c. On the displayed page, locate the target job and click **Edit** in the **Operation** column.

d. On the page displayed, select the latest Flink version. Flink 1.15 is recommended.

For the syntax of Flink 1.15, see **Flink OpenSource SQL 1.15 Usage**.

Announcement published on: July 6, 2023

1.4 EOS Announcement for DLI Spark 2.3.2

Description

Huawei Cloud initiated an end of service (EOS) for DLI Spark 2.3.2 at 00:00:00 (GMT+08:00) on December 31, 2023.

Impact

After the EOS, no technical support will be provided for DLI Spark 2.3.2. You are advised to select the Spark engine of the latest version when executing jobs. DLI Spark 3.3.1 is recommended.

For jobs that are using DLI Spark 2.3.2, switch to the Spark engine of the latest version as soon as possible. Otherwise, no technical support will be provided if an error occurs during job execution.

If you have any questions or suggestions, please **submit a service ticket** or call us on +86-4000-955-988 or +86-950-808.

FAQ

- **How does the EOS affect the jobs that are using DLI Spark 2.3.2?**

  If a queue is created after the EOS of Spark 2.3.2, the compute engine that has reached EOS cannot be selected during job execution.

  Historical queues can still use Spark 2.3.2 to execute jobs. However, if an error occurs during job execution, no technical support is provided. Replace the compute engine with a new version as soon as possible.
- **Which version can be used as a replacement after the EOS?**

  DLI Spark 3.3.1 is recommended.
- **What are the advantages of DLI Spark 3.3.1?**

Table 1-3 Advantages of Spark 3.3.1

Feature	Description
Native performance acceleration	Improved the performance of Spark query statements.
Metadata access performance improvement	Improved Spark's metadata access performance for handling big data and enhanced data processing efficiency.

Feature	Description
Improving the performance of OBS Committer when writing small files	Improved the performance of Object Storage Service (OBS) when writing small files, improving data transfer efficiency.
Dynamic executor shuffle data optimization	Improved the stability of resource scaling and cleaned up Executors when shuffle files are no longer needed.
Merging small files	If a large number of small files are generated during SQL execution, job execution and table query will take a long time. In this case, you are advised to merge small files. Merge small files by referring to **How Do I Merge Small Files?**
Modifying column comments of non-partitioned or partitioned tables	You can modify the column comments of non-partitioned or partitioned tables.
Collecting statistics on the CPU usage of SQL jobs	You can view the total CPU used on the console.
Viewing Spark logs of container clusters	You need to view logs in the container.
Dynamic UDF loading (OBT)	The UDF takes effect without restarting the queue.
Supporting flame graphs on the Spark UI	Flame graphs can be created on the Spark UI.
Optimizing the query performance of the **NOT IN** statement for SQL jobs	The query performance of the **NOT IN** statement is improved.
Optimizing the query performance of the **Multi-INSERT** statement	The query performance of the **Multi-INSERT** statement is improved.

- **Does the upgrade affect the DLI resource price?**

  DLI bills you based on the amount of compute and storage resources consumed by jobs, regardless of the compute engine version.
- **How do I upgrade DLI Spark to version 2.4.5?**

a. On the DLI management console, buy an elastic resource pool and create queues within the pool to provide compute resources required for job execution.
b. In the navigation pane on the left, choose **Job Management** > **Spark Jobs**. On the displayed page, click **Create Job** in the upper right corner.
c. On the displayed page, locate the target job and click **Edit** in the **Operation** column.
d. On the page displayed, select the latest Spark version. Spark 3.3.1 is recommended.

Announcement published on: July 6, 2023

1.5 EOS Announcement for DLI Flink 1.7

Description

Huawei Cloud initiated an end of service (EOS) for DLI Flink 1.7 at 00:00:00 (GMT +08:00) on December 31, 2022.

Impact

After the EOS, no technical support will be provided for DLI Flink 1.7. You are advised to select the Flink engine of the latest version when executing jobs. DLI Flink 1.15 is recommended.

For jobs that are using Flink 1.7, switch to the Flink engine of the latest version as soon as possible. Otherwise, no technical support will be provided if an error occurs during job execution.

If you have any questions or suggestions, please **submit a service ticket** or call us on +86-4000-955-988 or +86-950-808.

FAQ

- **Which functions of Flink 1.7 will not be evolved?**
  - The Flink Edge SQL function will no longer be supported for edge job processing after Flink 1.7 EOS, and subsequent versions of Flink will not support it either.
  - Similarly, the sensitive variable function will no longer be supported after Flink 1.7 EOS, and subsequent versions will not support it either.
- **How does the EOS affect the jobs that are using Flink 1.7?**

  If a queue is created after the EOS of Flink 1.7, the compute engine that has reached EOS cannot be selected during job execution.

  If you encounter any errors when Flink 1.7 is used to execute jobs on historical queues, note that this version will no longer receive any technical support. It is recommended that you switch to a later version of the compute engine as soon as possible.
- **Which version can be used as a replacement after the EOS?**

  DLI Flink 1.15 is recommended.

- **What are the advantages of Flink 1.12?**

  Flink 1.12 supports DataGen, GaussDB(DWS), JDBC, MySQL CDC, Postgres CDC, Redis, Upsert Kafka, and HBase source tables.

  For more advantages, see **Flink 1.12 Upgrade Guide**.

- **Does the upgrade of Flink affect the DLI resource price?**

  DLI bills you based on the amount of compute and storage resources consumed by jobs, regardless of the compute engine version.

- **How do I upgrade Flink to version 1.12?**

  a. On the DLI management console, buy an elastic resource pool and create queues within the pool to provide compute resources required for job execution.

  b. Log in to the DLI management console. In the navigation pane on the left, choose **Job Management** > **Flink Jobs**.

  c. On the displayed page, locate the target job and click **Edit** in the **Operation** column.

  d. On the page displayed, select the latest Flink version. Flink 1.15 is recommended.

Announcement published on: July 6, 2023

2 Version Support Bulletin

2.1 Lifecycle of DLI Compute Engine Versions

Version Description

DLI compute engine version is in *Compute engine name x.y.z* format. *Compute engine name* can be **Flink** or **Spark**. **Figure 2-1** describes the version.

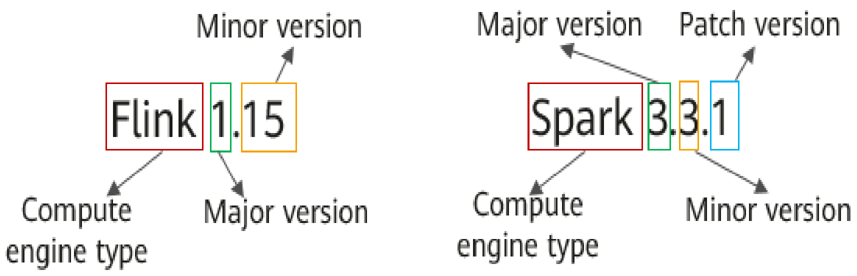


Figure 2-1 DLI compute engine version description

Version Support

- Recommended Flink version: Flink 1.15
- Recommended Spark version: Spark 3.3.1

NOTE

You are advised not to use Spark/Flink engines of different versions for a long time.

- Doing so can lead to code incompatibility, which can negatively impact job execution efficiency.
- Doing so may result in job execution failures due to conflicts in dependencies. Jobs rely on specific versions of libraries or components.

Lifecycle of a Compute Engine Version

Table 2-1 lists the lifecycle of DLI compute engine versions, based on which you can plan your version update pace.

Table 2-1 Lifecycle of DLI compute engine versions

Compute Engine Type	Version	Status	EOM Date	EOS Date
Flink	DLI Flink 1.15	Released	June 30, 2025	June 30, 2026
	DLI Flink 1.12	EOS	December 31, 2023	December 31, 2024
	DLI Flink 1.11	EOS	June 30, 2022	December 31, 2023
	DLI Flink 1.10	EOS	June 30, 2022	December 31, 2023
	DLI Flink 1.7	EOS	December 31, 2021	December 31, 2022
Spark	DLI Spark 3.3.1	Released	June 30, 2025	June 30, 2026
	DLI Spark 3.1.1	EOS	December 31, 2023	December 31, 2024
	DLI Spark 2.4.5	EOS	December 31, 2023	December 31, 2024
	DLI Spark 2.3.2	EOS	June 30, 2022	December 31, 2023

NOTE

- End of Marketing (EOM): indicates that the sales of this version are stopped. Any new purchases of resources will no longer support the engine version that has reached EOM.
- End of Service & Support (EOS): Services of this version are stopped. You are advised to use the engine of the latest version when running jobs. After this date, Huawei Cloud will no longer provide any technical support for the software version.

2.2 What's New in Flink 1.15

DLI complies with the release consistency of the open source Flink compute engine. This section describes the updates in Flink 1.15.

For details about Flink 1.15, see **Release Notes - Flink Jar 1.15** and **Flink OpenSource SQL 1.15 Usage**.

Flink 1.15 Release Date

Version	Release Date	Status	EOM Date	EOS Date
DLI Flink 1.15	June 2023	Released	June 30, 2025	June 30, 2026

For more version support information, see **Lifecycle of DLI Compute Engine Versions**.

Flink 1.15 Description

- The syntax design of Flink 1.15 has been improved to achieve higher compatibility and consistency with mainstream open-source technology standards.
- Flink 1.15 has added support for new connectors such as Hive and Hudi.
- For synchronous data migration scenarios in Flink 1.15, DataArts Studio's DataArts Migration is recommended.
- Flink 1.15 now supports integration with DEW-CSMS secret management, providing a privacy protection solution.
- Flink 1.15 supports minimal submission of Flink Jar jobs.

  NOTE

  Minimal submission means Flink only submits the necessary job dependencies, not the entire Flink environment. By setting the scope of non-Connector Flink dependencies (starting with **flink-**) and third-party libraries (like Hadoop, Hive, Hudi, and MySQL-CDC) to **provided**, you ensure these dependencies are excluded from the Jar job, avoiding conflicts with Flink core dependencies.

  - Only Flink 1.15 supports minimal submission of Flink Jar jobs. Enable this by configuring **flink.dli.job.jar.minimize-submission.enabled=true** in the runtime optimization parameters.
  - For Flink-related dependencies, use the **provided** scope by adding **<scope>provided</scope>** in the dependencies, especially for non-Connector dependencies under the **org.apache.flink** group starting with **flink-**.
  - For dependencies related to Hadoop, Hive, Hudi, and MySQL-CDC, also use the **provided** scope by adding **<scope>provided</scope>** in the dependencies.
  - In the Flink source code, only methods marked with **@Public** or **@PublicEvolving** are intended for user invocation. DLI guarantees compatibility with these methods.

2.3 What's New in Flink 1.12

DLI complies with the release consistency of the open source Flink compute engine. This section describes the updates in Flink 1.12.

For more information about Flink 1.12, see **Release Notes - Flink 1.12**.

Flink 1.12 Release Date

Version	Release Date	Status	EOM Date	EOS Date
DLI Flink 1.12	December 2021	EOS	December 31, 2023	December 31, 2024

For more version support information, see **Lifecycle of DLI Compute Engine Versions**.

Flink 1.12 Description

- Added support for DataGen, GaussDB(DWS), JDBC, MySQL CDC, Postgres CDC, Redis, Upsert Kafka, and HBase source tables.
- Added support for the merge of small files.
- Added support for Redis and RDS dimension tables.

2.4 What's New in Spark 3.3.1

DLI complies with the release consistency of the open source Spark compute engine. This section describes the updates in Spark 3.3.1.

For more information about Spark 3.3.1, see **Spark Release Notes**.

Spark 3.3.1 Release Date

Version	Release Date	Status	EOM Date	EOS Date
DLI Spark 3.3.1	June 2023	Released	June 30, 2025	June 30, 2026

For more version support information, see **Lifecycle of DLI Compute Engine Versions**.

Spark 3.3.1 Description

Table 2-2 lists the main features of Spark 3.3.1.

For more information on new features and performance optimizations, see **Release Notes - Spark 3.3.1**.

Table 2-2 Advantages of Spark 3.3.1

Feature	Description
Native performance acceleration	Improved the performance of Spark query statements.
Metadata access performance improvement	Improved Spark's metadata access performance for handling big data and enhanced data processing efficiency.
Improving the performance of OBS Committer when writing small files	Improved the performance of Object Storage Service (OBS) when writing small files, improving data transfer efficiency.
Dynamic executor shuffle data optimization	Improved the stability of resource scaling and cleaned up Executors when shuffle files are no longer needed.
Merging small files	If a large number of small files are generated during SQL execution, job execution and table query will take a long time. In this case, you are advised to merge small files. Merge small files by referring to **How Do I Merge Small Files?**
Modifying column comments of non-partitioned or partitioned tables	You can modify the column comments of non-partitioned or partitioned tables.
Collecting statistics on the CPU usage of SQL jobs	You can view the total CPU used on the console.
Viewing Spark logs of container clusters	You need to view logs in the container.
Dynamic UDF loading (OBT)	The UDF takes effect without restarting the queue.
Supporting flame graphs on the Spark UI	Flame graphs can be created on the Spark UI.
Optimizing the query performance of the **NOT IN** statement for SQL jobs	The query performance of the **NOT IN** statement is improved.
Optimizing the query performance of the **Multi-INSERT** statement	The query performance of the **Multi-INSERT** statement is improved.

2.5 What's New in Spark 3.1.1

DLI complies with the release consistency of the open source Spark compute engine. This section describes the updates in Spark 3.1.1.

For more information about Spark 3.1.1, see **Spark Release Notes**.

Spark 3.1.1 Release Date

Version	Release Date	Status	EOM Date	EOS Date
DLI Spark 3.1.1	December 2021	EOS	December 31, 2023	December 31, 2024

For more version support information, see **Lifecycle of DLI Compute Engine Versions**.

Spark 3.1.1 Description

The following lists the main features of Spark 3.1.1.

For more new features, see **Release Notes - Spark 3.1.1**.

- [SPARK-33050]: Upgraded Apache ORC to version 1.5.12.
- [SPARK-33092]: Improved subexpression elimination.
- [SPARK-33480]: Added support for the char/varchar data type.
- [SPARK-32302]: Optimized the pushdown of some predicates.
- [SPARK-30648]: Added support for the pushdown of predicates in JSON datasource tables.
- [SPARK-32346]: Added support for the pushdown of predicates in Avro datasource tables.
- [SPARK-32461]: Optimized the Shuffle Hash Join algorithm.
- [SPARK-32272]: Added the SQL-standard command **SET TIME ZONE**.
- [SPARK-21492]: Fixed memory leak caused by the sort-merge join algorithm.
- [SPARK-27812]: Upgraded the Kubernetes client to version 4.6.1.

NOTE

DLI does not support built-in geospatial query functions since Spark 3.*x*.

2.6 What's New in Spark 2.4.5

DLI complies with the release consistency of the open source Spark compute engine. This document describes the updates in Spark 2.4.5.

For more information about Spark 2.4.5, see **Spark Release Notes**.

Spark 2.4.5 Release Date

Version	Release Date	Status	EOM Date	EOS Date
DLI Spark 2.4.5	December 2021	EOS	December 31, 2023	December 31, 2024

For more version support information, see **Lifecycle of DLI Compute Engine Versions**.

Spark 2.4.5 Description

Table 2-3 lists the main features of Spark 2.4.5.

For more new features, see **Release Notes - Spark 2.4.5**.

Table 2-3 Advantages of Spark 2.4.5

Feature	Description
Merging small files	If a large number of small files are generated during SQL execution, job execution and table query will take a long time. In this case, you are advised to merge small files. Merge small files by referring to **How Do I Merge Small Files?**
Modifying column comments of non-partitioned or partitioned tables	You can modify the column comments of non-partitioned or partitioned tables.
Collecting statistics on the CPU usage of SQL jobs	You can view the total CPU used on the console.
Viewing Spark logs of container clusters	You need to view logs in the container.
Dynamic UDF loading (OBT)	The UDF takes effect without restarting the queue.
Supporting flame graphs on the Spark UI	Flame graphs can be created on the Spark UI.
Optimizing the query performance of the **NOT IN** statement for SQL jobs	The query performance of the **NOT IN** statement is improved.

Feature	Description
Optimizing the query performance of the **Multi-INSERT** statement	The query performance of the **Multi-INSERT** statement is improved.

2.7 Differences Between Spark 2.4.x and Spark 3.3.x

2.7.1 Differences in SQL Queues Between Spark 2.4.x and Spark 3.3.x

DLI has summarized the differences in SQL queues between Spark 2.4.*x* and Spark 3.3.*x* to help you understand the impact of upgrading the Spark version on jobs running in the SQL queues with the new engine.

Difference in the Return Type of the histogram_numeric Function

- **Explanation:**

  The **histogram_numeric** function in Spark SQL returns an array of structs (x, y), where the type of x varies between different engine versions.

  – **Spark 2.4.*x***: In Spark 3.2 or earlier, *x* is of type double.

  – **Spark 3.3.*x***: The type of *x* is equal to the input value type of the function.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; related usages need to be adapted.
- **Sample code**:

  Prepare data:

```
create table test_histogram_numeric(val int);
INSERT INTO test_histogram_numeric VALUES(1),(5),(8),(6),(7),(9),(8),(9);
```

  Execute SQL:

```
select histogram_numeric(val,3) from test_histogram_numeric;
```

  – Spark 2.4.5

```
[{"x":1.0,"y":1.0},{"x":5.5,"y":2.0},{"x":8.200000000000001,"y":5.0}]
```

  – Spark 3.3.1

```
[{"x":1,"y":1.0},{"x":5,"y":2.0},{"x":8,"y":5.0}]
```

Spark 3.3.x No Longer Supports Using "0$" to Specify the First Argument

- **Explanation:**

  In **format_string(strfmt, obj, ...)** and **printf(strfmt, obj, ...)**, **strfmt** will no longer support using **0$** to specify the first argument; the first argument should always be referenced by **1$** when using argument indexing to indicate the position of the argument in the parameter list.

  – **Spark 2.4.*x***: Both **%0** and **%1** can represent the first argument.

  – **Spark 3.3.*x***: **%0** is no longer supported.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; usages involving **%0** need to be modified to adapt to Spark 3.3.*x*.

- **Sample code 1**:

  Execute SQL:

  ```
  SELECT format_string('Hello, %0$s! I\'m %1$s!', 'Alice', 'Lilei');
  ```

  – Spark 2.4.5
    ```
    Hello, Alice! I'm Alice!
    ```
  – Spark 3.3.1
    ```
    DLI.0005: The value of parameter(s) 'strfmt' in `format_string` is invalid: expects %1$, %2$ and
    so on, but got %0$.
    ```

- **Sample code 2**:

  Execute SQL:

  ```
  SELECT format_string('Hello, %1$s! I\'m %2$s!', 'Alice', 'Lilei');
  ```

  – Spark 2.4.5
    ```
    Hello, Alice! I'm Lilei!
    ```
  – Spark 3.3.1
    ```
    Hello, Alice! I'm Lilei!
    ```

Spark 3.3.x Empty String Without Quotes

- **Explanation:**

  By default, in the CSV data source, empty strings are represented as **""** in Spark 2.4.5. After upgrading to Spark 3.3.1, empty strings have no quotes.

  – **Spark 2.4.***x*: Empty strings in the CSV data source are represented as **""**.

  – **Spark 3.3.***x*: Empty strings in the CSV data source have no quotes.

    To restore the format of Spark 2.4.*x* in Spark 3.3.*x*, you can set **spark.sql.legacy.nullValueWrittenAsQuotedEmptyStringCsv** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the storage format of null values in exported ORC files will be different.

- **Sample code**:

  Prepare data:

  ```
  create table test_null(id int,name string) stored as parquet;
  insert into test_null values(1,null);
  ```

  Export a CSV file and check the file content:

  – Spark 2.4.5
    ```
    1,""
    ```
  – Spark 3.3.1
    ```
    1,
    ```

Different Return Results of the describe function

- **Explanation:**

  If the function does not exist, **describe function** will fail.

  – **Spark 2.4.***x*: The **DESCRIBE** function can still run and print **Function: func_name not found**.

  – **Spark 3.3.***x*: The error message changes to failure if the function does not exist.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the return information of the **describe function** related APIs is different.

- **Sample code**:

  Execute SQL:

  ```
  describe function dli_no (dli_no does not exist)
  ```

  – Spark 2.4.5

    Successfully executed, **function_desc** content:

    ```
    Function:func_name not found
    ```

  – Spark 3.3.1

    Execution failed, DLI.0005:

    ```
    Undefined function: dli_no...
    ```

Clear Indication That Specified External Table Property is Not Supported

- **Explanation:**

  The **external** property of the table becomes reserved. If the external property is specified, certain commands will fail.

  – **Spark 2.4.***x*: Commands succeed when specifying the **external** property via **CREATE TABLE ... TBLPROPERTIES** and **ALTER TABLE ... SET TBLPROPERTIES**, but the **external** property is silently ignored, and the table remains a managed table.

  – **Spark 3.3.***x*:

    Commands will fail when specifying the **external** property via **CREATE TABLE ... TBLPROPERTIES** and **ALTER TABLE ... SET TBLPROPERTIES**.

    To restore the Spark 2.4.*x* usage in Spark 3.3.*x*, set **spark.sql.legacy.notReserveProperties** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; related usages need to be adapted.

- **Sample code**:

  Execute SQL:

  ```
  CREATE TABLE test_external(id INT,name STRING) TBLPROPERTIES('external'=true);
  ```

  – Spark 2.4.5

    Successfully executed.

  – Spark 3.3.1

    ```
    DLI.0005: The feature is not supported: external is a reserved table property, please use CREATE
    EXTERNAL TABLE.
    ```

New Support for Parsing Strings of "+Infinity", "+INF", and "-INF" Types

- **Explanation:**

  – **Spark 2.4.***x*: When reading values from JSON properties defined as **FloatType** or **DoubleType**, Spark 2.4.*x* only supports parsing **Infinity** and **-Infinity**.

  – **Spark 3.3.***x*: In addition to supporting **Infinity** and **-Infinity**, Spark 3.3.*x* also supports parsing strings of **+Infinity**, **+INF**, and **-INF**.

- **Is there any impact on jobs after the engine version upgrade?**

Function enhancement, no impact.

Default Configuration spark.sql.adaptive.enabled = true

- **Explanation:**
  - **Spark 2.4.*x***: In Spark 2.4.*x*, the default value of the **spark.sql.adaptive.enabled** configuration item is **false**, meaning that the Adaptive Query Execution (AQE) feature is disabled.
  - **Spark 3.3.*x***: Starting from Spark 3.3.*x*-320, AQE is enabled by default, that is, **spark.sql.adaptive.enabled** is set to **true**.
- **Is there any impact on jobs after the engine version upgrade?**

  DLI functionality is enhanced, and the default value of **spark.sql.adaptive.enabled** has changed.

Change in the Schema of SHOW TABLES Output

- **Explanation:**

  The schema of the **SHOW TABLES** output changes from **database: string** to **namespace: string**.
  - **Spark 2.4.*x***: The schema of the **SHOW TABLES** output is **database: string**.
  - **Spark 3.3.*x***:

    The schema of the **SHOW TABLES** output changes from **database: string** to **namespace: string**.

    For the built-in catalog, the **namespace** field is named **database**; for the v2 catalog, there is no **isTemporary** field.

    To revert to the style of Spark 2.4.*x* in Spark 3.3.*x*, set **spark.sql.legacy.keepCommandOutputSchema** to **true**.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; check the usages related to **SHOW TABLES** in jobs and adapt them to meet the new version's usage requirements.
- **Sample code**:

  Execute SQL:

```
show tables;
```

  - Spark 2.4.5

```
database    tableName    isTemporary
db1         table1        false
```

  - Spark 3.3.1

```
namespace    tableName    isTemporary
db1           table1        false
```

Schema Change in SHOW TABLE EXTENDED Output

- **Explanation:**

  The schema of the **SHOW TABLE EXTENDED** output changes from **database: string** to **namespace: string**.
  - **Spark 2.4.*x***: The schema of the **SHOW TABLE EXTENDED** output is **database: string**.

- **Spark 3.3.*x***:
- The schema of the **SHOW TABLE EXTENDED** output changes from **database: string** to **namespace: string**.

  For the built-in catalog, the **namespace** field is named **database**; there is no change for the v2 catalog.

  To revert to the style of Spark 2.4.*x* in Spark 3.3.*x*, set **spark.sql.legacy.keepCommandOutputSchema** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; check the usages related to **SHOW TABLES** in jobs and adapt them to meet the new version's usage requirements.

- **Sample code**:

  Execute SQL:

```
show table extended like 'table%';
```

  - Spark 2.4.5

```
database    tableName    isTemporary    information
db1         table1       false          Database:db1...
```

  - Spark 3.3.1

```
namespace    tableName    isTemporary    information
db1          table1       false          Database:db1...
```

Impact of Table Refresh on Dependent Item Cache

- **Explanation:**

  After upgrading to Spark 3.3.*x*, table refresh will clear the table's cache data but keep the dependent item cache.

  - **Spark 2.4.*x***: In Spark 2.4.*x*, when performing a table refresh operation (for example, **REFRESH TABLE**), the cache data of dependent items (for example, views) is not retained.

```
ALTER TABLE .. ADD PARTITION
ALTER TABLE .. RENAME PARTITION
ALTER TABLE .. DROP PARTITION
ALTER TABLE .. RECOVER PARTITIONS
MSCK REPAIR TABLE
LOAD DATA
REFRESH TABLE
TRUNCATE TABLE
spark.catalog.refreshTable
```

  - Spark 3.3.*x*: After upgrading to Spark 3.3.*x*, table refresh will clear the table's cache data but keep the dependent item cache.

- **Is there any impact on jobs after the engine version upgrade?**

  The upgraded engine version will increase the cache data of the original dependencies.

Impact of Table Refresh on Other Cached Operations Dependent on the Table

- **Explanation:**

  - **Spark 2.4.*x***: In Spark 2.4.*x*, refreshing a table will trigger the uncache operation for all other caches referencing the table only if the table itself is cached.

   - **Spark 3.3.*x***: After upgrading to the new engine version, refreshing the table will trigger the uncache operation for other caches dependent on the table, regardless of whether the table itself is cached.
- **Is there any impact on jobs after the engine version upgrade?**

  DLI functionality is enhanced to ensure that the table refresh operation can affect the cache, improving program robustness.

New Support for Using Typed Literals in ADD PARTITION

- **Explanation:**
   - **Spark 2.4.*x***:

     In Spark 2.4.*x*, using typed literals (for example, **date'2020-01-01'**) in **ADD PARTITION** will parse the partition value as a string **date'2020-01-01'**, resulting in an invalid date value and adding a partition with a null value.

     The correct approach is to use a string value, such as **ADD PARTITION(dt = '2020-01-01')**.
   - **Spark 3.3.*x***: In Spark 3.3.*x*, partition operations support using typed literals, supporting **ADD PARTITION(dt = date'2020-01-01')** and correctly parsing the partition value as a date type instead of a string.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the handling of typed literals in **ADD PARTITION** has changed.
- **Sample code**:

  Prepare data:

```
create table test_part_type (id int,name string,pt date) PARTITIONED by (pt);
insert into test_part_type partition (pt = '2021-01-01') select 1,'name1';
insert into test_part_type partition (pt = date'2021-01-01') select 1,'name1';
```

  Execute SQL:

```
select id,name,pt from test_part_type;
(Set the parameter spark.sql.forcePartitionPredicatesOnPartitionedTable.enabled to false.)
```

   - Spark 2.4.5

```
1 name1 2021-01-01
1 name1
```

   - Spark 3.3.1

```
1 name1 2021-01-01
1 name1 2021-01-01
```

Mapping Type Change of DayTimeIntervalType to Duration

- **Explanation:**

  In the ArrowWriter and ArrowColumnVector developer APIs, starting from Spark 3.3.*x*, the **DayTimeIntervalType** in Spark SQL is mapped to Apache Arrow's **Duration** type.
   - **Spark 2.4.*x***: **DayTimeIntervalType** is mapped to Apache Arrow's **Interval** type.
   - **Spark 3.3.*x***: **DayTimeIntervalType** is mapped to Apache Arrow's **Duration** type.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the mapping type of **DayTimeIntervalType** has changed.

Type Change in the Return Result of Date Difference

- **Explanation:**

  The date subtraction expression (e.g., date1 – date2) returns a **DayTimeIntervalType** value.

  – **Spark 2.4.*x***: Returns **CalendarIntervalType**.

  – **Spark 3.3.*x***: Returns **DayTimeIntervalType**.

    To restore the previous behavior, set **spark.sql.legacy.interval.enabled** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the default type of the date difference return result has changed.

Mapping Type Change of Unit-to-Unit Interval

- **Explanation:**

  – **Spark 2.4.*x***: In Spark 2.4.*x*, unit-to-unit intervals (e.g., **INTERVAL '1-1' YEAR TO MONTH**) and unit list intervals (e.g., **INTERVAL '3' DAYS '1' HOUR**) are converted to **CalendarIntervalType**.

  – **Spark 3.3.*x***: In Spark 3.3.*x*, unit-to-unit intervals and unit list intervals are converted to ANSI interval types: **YearMonthIntervalType** or **DayTimeIntervalType**.

    To restore the mapping type to that before Spark 2.4.*x* in Spark 3.3.*x*, set the configuration item **spark.sql.legacy.interval.enabled** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the mapped data type has changed.

Return Value Type Change in timestamps Subtraction Expression

- **Explanation:**

  – **Spark 2.4.*x***: In Spark 2.4.*x*, the timestamps subtraction expression (for example, **select timestamp'2021-03-31 23:48:00' – timestamp'2021-01-01 00:00:00'**) returns a **CalendarIntervalType** value.

  – **Spark 3.3.*x***: In Spark 3.3.*x*, the timestamps subtraction expression (for example, **select timestamp'2021-03-31 23:48:00' – timestamp'2021-01-01 00:00:00'**) returns a **DayTimeIntervalType** value.

    To restore the mapping type to that before Spark 2.4.*x* in Spark 3.3.*x*, set **spark.sql.legacy.interval.enabled** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the mapped data type has changed.

Mixed Use of Year-Month and Day-Time Fields No Longer Supported

- **Explanation:**

  – **Spark 2.4.*x***: Unit list interval literals can mix year-month fields (**YEAR** and **MONTH**) and day-time fields (**WEEK**, **DAY**, ..., **MICROSECOND**).

- **Spark 3.3.*x***: Unit list interval literals cannot mix year-month fields (**YEAR** and **MONTH**) and day-time fields (**WEEK**, **DAY**, ..., **MICROSECOND**). Invalid input is indicated.

  To restore the usage to that before Spark 2.4.*x* in Spark 3.3.*x*, set **spark.sql.legacy.interval.enabled** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact.

Reserved Properties Cannot Be Used in CREATE TABLE .. LIKE .. Command

- **Explanation:**

  Reserved properties cannot be used in the **CREATE TABLE .. LIKE ..** command.

  - **Spark 2.4.*x***: In Spark 2.4.*x*, the **CREATE TABLE .. LIKE ..** command can use reserved properties.

    For example, **TBLPROPERTIES('location'='/tmp')** does not change the table location but creates an invalid property.

  - **Spark 3.3.*x***: In Spark 3.3.*x*, the **CREATE TABLE .. LIKE ..** command cannot use reserved properties.

    For example, using **TBLPROPERTIES('location'='/tmp')** or **TBLPROPERTIES('owner'='yao')** will fail.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact.

- **Sample code 1**:

  Prepare data:

  ```
  CREATE TABLE test0(id int, name string);
  CREATE TABLE test_like_properties LIKE test0 LOCATION 'obs://bucket1/test/test_like_properties';
  ```

  Execute SQL:

  ```
  DESCRIBE FORMATTED test_like_properties;
  ```

  - Spark 2.4.5

    The location is properly displayed.

  - Spark 3.3.1

    The location is properly displayed.

- **Sample code 2**:

  Prepare data:

  ```
  CREATE TABLE test_like_properties0(id int) using parquet LOCATION 'obs://bucket1/dbgms/
  test_like_properties0';
  CREATE TABLE test_like_properties1 like test_like_properties0 tblproperties('location'='obs://bucket1/
  dbgms/test_like_properties1');
  ```

  Execute SQL:

  ```
  DESCRIBE FORMATTED test_like_properties1;
  ```

  - Spark 2.4.5

    ```
    DLI.0005:
    mismatched input 'tblproperties' expecting {<EOF>, 'LOCATION'}
    ```

  - Spark 3.3.1

    ```
    The feature is not supported: location is a reserved table property, please use the LOCATION
    clause to specify it.
    ```

Failure to Create a View with Auto-Generated Aliases

- **Explanation:**
  - **Spark 2.4.*x***: If the statement contains an auto-generated alias, it will execute normally without any prompt.
  - **Spark 3.3.*x***: If the statement contains an auto-generated alias, creating/changing the view will fail.

    To restore the usage to that before Spark 2.4.*x* in Spark 3.3.*x*, set **spark.sql.legacy.allowAutoGeneratedAliasForView** to **true**.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact.
- **Sample code**:

  Prepare data:

  ```
  create table test_view_alis(id1 int,id2 int);
  INSERT INTO test_view_alis VALUES(1,2);
  ```

  Execute SQL:

  ```
  create view view_alis as select id1 + id2 from test_view_alis;
  ```

  - Spark 2.4.5

    Successfully executed.
  - Spark 3.3.1

    ```
    Error
    Not allowed to create a permanent view `view_alis` without explicitly assigning an alias for
    expression (id1 + id2)
    ```

    If the following parameter is added in Spark 3.3.1, the SQL will execute successfully:

    ```
    spark.sql.legacy.allowAutoGeneratedAliasForView = true
    ```

Change in Return Value Type After Adding/Subtracting Time Field Intervals to/from Dates

- **Explanation:**

  The return type changes when adding/subtracting a time interval (for example, 12 hours) to/from a date-time field (for example, **date'2011-11-11'**).
  - **Spark 2.4.*x***: In Spark 2.4.*x*, when performing date arithmetic operations on JSON attributes defined as **FloatType** or **DoubleType**, such as **date'2011-11-11'** plus or minus a time interval (such as 12 hours), the return type is **DateType**.
  - **Spark 3.3.*x***: The return type changes to a timestamp (**TimestampType**) to maintain compatibility with Hive.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact.
- **Sample code**:

  Execute SQL:

  ```
  select date '2011-11-11' - interval 12 hour
  ```

  - Spark 2.4.5

    ```
    2011-11-10
    ```

– Spark 3.3.1
```
1320897600000
```

Support for Char/Varchar Types in Spark SQL

- **Explanation:**
  – **Spark 2.4.*x***: Spark SQL table columns do not support Char/Varchar types; when specified as **Char** or **Varchar**, they are forcibly converted to the **String** type.
  – **Spark 3.3.*x***: Spark SQL table columns support **CHAR/CHARACTER** and **VARCHAR** types.
- **Is there any impact on jobs after the engine version upgrade?**
  There is no impact.
- **Sample code**:
  Prepare data:
```
create table test_char(id int,name varchar(24),name2 char(24));
```
  Execute SQL:
```
show create table test_char;
```
  – Spark 2.4.5
```
create table `test_char`(`id` INT,`name` STRING,`name2` STRING)
ROW FORMAT...
```
  – Spark 3.3.1
```
create table test_char(id INT,name VARCHAR(24),name2 VARCHAR(24))
ROW FORMAT...
```

Different Query Syntax for Null Partitions

- **Explanation:**
  – **Spark 2.4.*x***:
    In Spark 3.0.1 or earlier, if the partition column is of type string, it is parsed as its text representation, such as the string "null".
    Querying null partitions using **part_col='null'**.
  – **Spark 3.3.*x***:
    PARTITION(col=null) always parses to null in partition specification, even if the partition column is of type string.
    Querying null partitions using **part_col is null**.
- **Is there any impact on jobs after the engine version upgrade?**
  There is an impact; queries for null partitions need to be adapted.
- **Sample code**:
  Prepare data:
```
CREATE TABLE test_part_null (col1 INT, p1 STRING) USING PARQUET PARTITIONED BY (p1);
INSERT INTO TABLE test_part_null PARTITION (p1 = null) SELECT 0;
```
  Execute SQL:
```
select * from test_part_null;
```
  – Spark 2.4.5
```
0 null
```
    Executing **select * from test_part_null where p1='null'** can find partition data in Spark 2.4.5.

- Spark 3.3.1

  ```
  0
  ```

  Executing **select * from test_part_null where p1 is null** can find data in Spark 3.3.1.

Different Handling of Partitioned Table Data

- **Explanation:**

  In datasource v1 partition external tables, non-UUID partition path data already exists.

  Performing the **insert overwrite partition** operation in Spark 3.3.*x* will clear previous non-UUID partition data, whereas Spark 2.4.*x* will not.

  - **Spark 2.4.*x***:

    Retains data under non-UUID partition paths.

  - **Spark 3.3.*x***:

    Deletes data under non-UUID partition paths.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; it will clean up dirty data.

- **Sample code**:

  Prepare data:

  Create a directory named **pt=pt1** under **obs://bucket1/test/overwrite_datasource** and import a Parquet data file into it.

  ```
  create table overwrite_datasource(id int,name string,pt string) using parquet PARTITIONED by(pt)
  LOCATION 'obs://bucket1/test/overwrite_datasource';
  SELECT * FROM overwrite_datasource1 where pt='pt1' Both versions do not query data.
  ```

  Execute SQL:

  ```
  insert OVERWRITE table overwrite_datasource partition(pt='pt1') values(2,'aa2');
  ```

  - Spark 2.4.5

    Retains the **pt=pt1** directory.

  - Spark 3.3.1

    Deletes the **pt=pt1** directory.

Retaining Quotes for Special Characters When Exporting CSV Files

- **Explanation:**
  - **Spark 2.4.*x***:

    When exporting CSV files in Spark 2.4.*x*, if field values contain special characters such as newline (\n) and carriage return (\r), and these special characters are surrounded by quotes (e.g., double quotes "), Spark automatically handles these quotes, omitting them in the exported CSV file.

    For example, the field value **"a\rb"** will not include quotes when exported.

  - **Spark 3.3.*x***:

    In Spark 3.3.*x*, the handling of exporting CSV files is optimized; if field values contain special characters and are surrounded by quotes, Spark retains these quotes in the final CSV file.

For example, the field value **"a\rb"** retains quotes when exported.

- **Is there any impact on jobs after the engine version upgrade?**

  No impact on query results, but it affects the export file format.

- **Sample code**:

  Prepare data:

  ```
  create table test_null2(str1 string,str2 string,str3 string,str4 string);
  insert into test_null2 select "a\rb", null, "1\n2", "ab";
  ```

  Execute SQL:

  ```
  SELECT * FROM test_null2;
  ```

  – Spark 2.4.5
    ```
    a b  1 2 ab
    ```
  – Spark 3.3.1
    ```
    a b  1 2 ab
    ```

  Export query results to OBS and check the CSV file content:

  – Spark 2.4.5
    ```
    a
    b,"","1
    2",ab
    ```
  – Spark 3.3.1
    ```
    "a
    b",,"1
    2",ab
    ```

New Support for Adaptive Skip Partial Agg Configuration

- **Explanation:**

  Spark 3.3.*x* introduces support for adaptive Skip partial aggregation. When partial aggregation is ineffective, it can be skipped to avoid additional performance overhead. Related parameters:

  – **spark.sql.aggregate.adaptivePartialAggregationEnabled**: controls whether to enable adaptive Skip partial aggregation. When set to **true**, Spark dynamically decides whether to skip partial aggregation based on runtime statistics.

  – **spark.sql.aggregate.adaptivePartialAggregationInterval**: configures the analysis interval, that is, after processing how many rows, Spark will analyze whether to skip partial aggregation.

  – **spark.sql.aggregate.adaptivePartialAggregationRatio**: Threshold to determine whether to skip, based on the ratio of Processed groups/Processed rows. If the ratio exceeds the configured threshold, Spark considers pre-aggregation ineffective and may choose to skip it to avoid further performance loss.

  During usage, the system first analyzes at intervals configured by **spark.sql.aggregate.adaptivePartialAggregationInterval**. When the processed rows reach the interval, it calculates Processed groups/Processed rows. If the ratio exceeds the threshold, it considers pre-aggregation ineffective and can directly skip it.

- **Is there any impact on jobs after the engine version upgrade?**

  Enhances DLI functionality.

New Support for Parallel Multi-Insert

- **Explanation:**

  Spark 3.3.*x* adds support for Parallel Multi-Insert. In scenarios with multi-insert SQL, where multiple tables are inserted in the same SQL, this type of SQL is serialized in open-source Spark, limiting performance. Spark 3.3.*x* introduces multi-insert parallelization optimization in DLI, allowing all inserts to be executed concurrently, improving performance.

  Enable the following features by setting them to **true** (default **false**):

  spark.sql.lazyExecutionForDDL.enabled=true

  spark.sql.parallelMultiInsert.enabled=true

- **Is there any impact on jobs after the engine version upgrade?**

  Enhances DLI functionality, improving the reliability of jobs with multi-insert parallelization features.

New Support for Enhance Reuse Exchange

- **Explanation:**

  Spark 3.3.*x* introduces support for Enhance Reuse Exchange. When the SQL plan includes reusable **sort merge join** conditions, setting **spark.sql.execution.enhanceReuseExchange.enabled** to **true** allows reuse of SMJ plan nodes.

  Enable the following features by setting them to **true** (default **false**):

  spark.sql.execution.enhanceReuseExchange.enabled=true

- **Is there any impact on jobs after the engine version upgrade?**

  Enhances DLI functionality.

Difference in Reading TIMESTAMP Fields

- **Explanation:**

  Differences in reading TIMESTAMP fields for the Asia/Shanghai time zone. For values before **1900-01-01 08:05:43**, values written by Spark 2.4.5 and read by Spark 3.3.1 differ from values read by Spark 2.4.5.

  - **Spark 2.4.*x***:

    For the Asia/Shanghai time zone, 1900-01-01 00:00:00, written in Spark 2.4.5 and read by Spark 2.4.5 returns **-2209017600000**.

  - **Spark 3.3.*x***:

    For the Asia/Shanghai time zone, 1900-01-01 00:00:00, written in Spark 2.4.5, read by Spark 3.3.1 with **spark.sql.parquet.int96RebaseModeInRead=LEGACY** returns **-2209017943000**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the usage of TIMESTAMP fields needs to be evaluated.

- **Sample code**:

  Configure in the SQL interface:

  ```
  spark.sql.session.timeZone=Asia/Shanghai
  ```

  - Spark 2.4.5

```
create table parquet_timestamp_test (id int, col0 string, col1 timestamp) using parquet;
insert into parquet_timestamp_test values (1, "245", "1900-01-01 00:00:00");
```

Execute SQL to read data:

```
select * from parquet_timestamp_test;
```

Query results:

```
id   col0    col1
1    245     -2209017600000
```

– Spark 3.3.1

```
spark.sql.parquet.int96RebaseModeInRead=LEGACY
```

Execute SQL to read data:

```
select * from parquet_timestamp_test;
```

Query results:

```
id   col0    col1
1    245     -2209017943000
```

- **Configuration description:**

  These configuration items specify how Spark handles **DATE** and **TIMESTAMP** type fields for specific times (times that are contentious between the proleptic Gregorian and Julian calendars). For example, in the Asia/Shanghai time zone, it refers to how values before 1900-01-01 08:05:43 are processed.

  – **Configuration items in a datasource Parquet table**

Table 2-4 Configuration items in a Spark 3.3.1 datasource Parquet table

Configuration Item	Default Value	Description
spark.sql.parquet.int96RebaseModeInRead	**LEGACY** (default for Spark SQL jobs)	Takes effect when reading INT96 type **TIMESTAMP** fields in Parquet files. ● **EXCEPTION**: Throws an error for specific times, causing the read operation to fail. ● **CORRECTED**: Reads the date/ timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/ timestamp from the traditional hybrid calendar (Julian + Gregorian) to the proleptic Gregorian calendar. This setting only takes effect when the write information for the Parquet file (e.g., Spark, Hive) is unknown.

Configuration Item	Default Value	Description
spark.sql.parquet.int96RebaseModeInWrite	**LEGACY** (default for Spark SQL jobs)	Takes effect when writing INT96 type **TIMESTAMP** fields in Parquet files. ● **EXCEPTION**: Throws an error for specific times, causing the write operation to fail. ● **CORRECTED**: Writes the date/ timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/ timestamp from the proleptic Gregorian calendar to the traditional hybrid calendar (Julian + Gregorian) when writing Parquet files.

Configuration Item	Default Value	Description
spark.sql.parquet.datetimeRebaseModeIn-Read	**LEGACY** (default for Spark SQL jobs)	Takes effect when reading **DATE**, **TIMESTAMP_MILLIS**, and **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the read operation to fail. ● **CORRECTED**: Reads the date/ timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/ timestamp from the traditional hybrid calendar (Julian + Gregorian) to the proleptic Gregorian calendar. This setting only takes effect when the write information for the Parquet file (e.g., Spark, Hive) is unknown.

Configuration Item	Default Value	Description
spark.sql.parquet.datetimeRebaseModeIn-Write	**LEGACY** (default for Spark SQL jobs)	Takes effect when writing **DATE**, **TIMESTAMP_MILLIS**, and **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the write operation to fail. ● **CORRECTED**: Writes the date/timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/timestamp from the proleptic Gregorian calendar to the traditional hybrid calendar (Julian + Gregorian) when writing Parquet files.

– **Configuration items in a datasource Avro table**

Table 2-5 Configuration items in a Spark 3.3.1 datasource Avro table

Configuration Item	Default Value	Description
spark.sql.avro.datetimeRebaseModeInRead	**LEGACY** (default for Spark SQL jobs)	Takes effect when reading **DATE**, **TIMESTAMP_MILLIS**, **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the read operation to fail. ● **CORRECTED**: Reads the date/ timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/ timestamp from the traditional hybrid calendar (Julian + Gregorian) to the proleptic Gregorian calendar. This setting only takes effect when the write information for the Avro file (e.g., Spark, Hive) is unknown.

Configuration Item	Default Value	Description
spark.sql.avro.datetimeRebaseModeInWrite	**LEGACY** (default for Spark SQL jobs)	Takes effect when writing **DATE**, **TIMESTAMP_MILLIS**, and **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the write operation to fail. ● **CORRECTED**: Writes the date/timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/timestamp from the proleptic Gregorian calendar to the traditional hybrid calendar (Julian + Gregorian) when writing Avro files.

Difference in from_unixtime Function

- **Explanation:**
  - **Spark 2.4.***x*:

    For the Asia/Shanghai time zone, **-2209017600** returns **1900-01-01 00:00:00**.
  - **Spark 3.3.***x*:

    For the Asia/Shanghai time zone, **-2209017943** returns **1900-01-01 00:00:00**.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; usage of this function needs to be checked.
- **Sample code**:

  Configure in the SQL interface:

  ```
  spark.sql.session.timeZone=Asia/Shanghai
  ```
  - Spark 2.4.5

    Execute SQL to read data:

    ```
    select from_unixtime(-2209017600);
    ```
    Query results:

```
 1900-01-01 00:00:00
```

– Spark 3.3.1

  Execute SQL to read data:

```
select from_unixtime(-2209017600);
```

  Query results:

```
1900-01-01 00:05:43
```

Difference in unix_timestamp Function

- **Explanation:**

  For values less than **1900-01-01 08:05:43** in the Asia/Shanghai time zone.

  – **Spark 2.4.***x*:

    For the Asia/Shanghai time zone, **1900-01-01 00:00:00** returns **-2209017600**.

  – **Spark 3.3.***x*:

    For the Asia/Shanghai time zone, **1900-01-01 00:00:00** returns **-2209017943**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; usage of this function needs to be checked.

- **Sample code**:

  Configure in the SQL interface:

```
spark.sql.session.timeZone=Asia/Shanghai
```

  – Spark 2.4.5

    Execute SQL to read data:

```
select unix_timestamp('1900-01-01 00:00:00');
```

    Query results:

```
 -2209017600
```

  – Spark 3.3.1

    Execute SQL to read data:

```
select unix_timestamp('1900-01-01 00:00:00');
```

    Query results

```
-2209017943
```

2.7.2 Differences in General-Purpose Queues Between Spark 2.4.x and Spark 3.3.x

DLI has summarized the differences in general-purpose queues between Spark 2.4.*x* and Spark 3.3.*x* to help you understand the impact of upgrading the Spark version on jobs running in the general queues with the new engine.

Log4j Dependency Updated from 1.x to 2.x

- **Explanation:**

  Log4j dependency is updated from 1.*x* to 2.*x*.

  – **Spark 2.4.***x*: Log4j dependency version 1.*x* (no longer supported by the community).

  - **Spark 3.3.*x***: Log4j dependency version 2.*x*.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact.

Spark 3.3.x Does Not Support v1 Tables

- **Explanation:**

  Spark 2.4.*x* supports datasource v1 and v2 tables. Spark 3.3.*x* does not support datasource v1 tables.

  For details, see **DLI Datasource V1 Table and Datasource V2 Table**.

  - **Spark 2.4.*x*** supports datasource v1 and v2 tables.
  - **Spark 3.3.*x*** does not support datasource v1 tables.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact. You are advised to migrate to v2 tables in Spark 2.4.5 before upgrading to Spark 3.3.1. For details, refer to the examples in **DLI Datasource V1 Table and Datasource V2 Table**.

Empty Input Splits Do Not Create Partitions by Default

- **Explanation:**
  - **Spark 2.4.*x***: Empty input splits create partitions by default.
  - **Spark 3.3.*x***: Empty input splits do not create partitions by default.

    When using Spark 3.3.*x*, **spark.hadoopRDD.ignoreEmptySplits** is set to **true**.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; it needs to be determined if partition names are used for service judgments.

Event Log Compression Format Set to zstd

- **Explanation:**

  In Spark 3.3.*x*, the default value for **spark.eventLog.compression.codec** is set to **zstd**. Spark will no longer use the value of **spark.io.compression.codec** for compressing event logs.

  - **Spark 2.4.*x***: Uses the value of **spark.io.compression.codec** for event log compression format.
  - **Spark 3.3.*x***: **spark.eventLog.compression.codec** is set to **zstd** by default.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the event log compression format changes.

Change in spark.launcher.childConectionTimeout Configuration

- **Explanation:**
  - **Spark 2.4.*x***: The configuration name is **spark.launcher.childConectionTimeout**.
  - **Spark 3.3.*x***: The configuration name is changed to **spark.launcher.childConnectionTimeout**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; configuration parameter names change.

Spark 3.3.x No Longer Supports Apache Mesos as a Resource Manager

- **Explanation:**
  - **Spark 2.4.***x*: Uses Apache Mesos as a resource manager.
  - **Spark 3.3.***x*: No longer supports using Apache Mesos as a resource manager.
- **Is there any impact on jobs after the engine version upgrade?**

  Functional enhancement. If you were using Mesos as a resource manager in Spark 2.4.*x*, you need to consider switching to another resource manager after upgrading to Spark 3.3.*x*.

Spark 3.3.x Deletes Kubernetes Driver When the Application Terminates Itself

- **Explanation**: Spark 3.3.*x* deletes the Kubernetes driver when the application terminates itself.
- **Is there any impact on jobs after the engine version upgrade?**

  Functional enhancement. After upgrading to Spark 3.3.*x*, this affects jobs that rely on Kubernetes as a resource manager. Spark 3.3.*x* automatically deletes the driver pod when the application terminates, which may affect resource management and cleanup processes.

Spark 3.3.x Supports Custom Kubernetes Schedulers

- **Explanation:**
  - **Spark 2.4.***x*: Does not support using a specified Kubernetes scheduler to manage resource allocation and scheduling for Spark jobs.
  - **Spark 3.3.***x*: Supports custom Kubernetes schedulers.
- **Is there any impact on jobs after the engine version upgrade?**

  Functional enhancement; supports custom schedulers for resource allocation and scheduling management.

Spark Converts Non-Nullable Schemas to Nullable

- **Explanation:**

  In Spark 2.4.*x*, when the user-specified schema contains non-nullable fields, Spark converts these non-nullable schemas to nullable.

  In Spark 3.3.*x*, Spark respects the nullability specified in the user schema, that is, if a field is defined as non-nullable, Spark retains this requirement and does not automatically convert it to a nullable field.
  - Spark 2.4.*x*: In Spark 2.4.*x*, when the user-specified schema contains non-nullable fields, Spark converts these non-nullable schemas to nullable.
  - **Spark 3.3.***x*: Does not automatically convert non-nullable fields to nullable.

    To revert to the behavior of Spark 2.4.*x* in Spark 3.3.*x*, set **spark.sql.legacy.respectNullabilityInTextDatasetConversion** to **true**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact.

- **Sample code**:

  Execute SQL:

  ```
  spark.read.schema(StructType(
  StructField("f1", LongType, nullable = false) ::
  StructField("f2", LongType, nullable = false) :: Nil)
  ).option("mode", "DROPMALFORMED").json(Seq("""{"f1": 1}""").toDS).show(false);
  ```

  – Spark 2.4.5

    ```
    |f1 |f2 |
    +---+---+
    |1  |0  |
    ```

  – Spark 3.3.1

    ```
    |f1 |f2  |
    +---+----+
    |1  |null|
    ```

Change in Spark Scala Version

- **Explanation:**

  The Spark Scala version changes.

  – **Spark 2.4.***x*: Uses Scala 2.11.

  – **Spark 3.3.***x*: Upgrades to Scala 2.12.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; jars need to be recompiled with the updated Scala version.

Change in Supported Python Versions for PySpark

- **Explanation:**

  The supported Python versions for PySpark change.

  – **Spark 2.4.***x*: PySpark supports Python 2.6+ to 3.7+.

  – **Spark 3.3.***x*: PySpark supports Python 3.6 or later.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact due to dependency version changes. You need to check whether this issue is involved.

Change in Supported Pandas Versions for PySpark

- **Explanation:**

  – **Spark 2.4.***x*: PySpark does not specify a Pandas version.

  – **Spark 3.3.***x*: From Spark 3.3.*x*, PySpark requires Pandas 0.23.2 or later to use Pandas-related functions like toPandas and createDataFrame from Pandas DataFrame.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact due to dependency version changes. You need to check whether this issue is involved.

Change in Supported PyArrow Versions for PySpark

- **Explanation:**

  - **Spark 2.4.*x***: PySpark does not specify a PyArrow version.
  - **Spark 3.3.*x***: From Spark 3.3.*x*, PySpark requires PyArrow 0.12.1 or later to use PyArrow-related functions like Pandas_udf and toPandas.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact due to dependency version changes. You need to check whether this issue is involved.

DataFrameWriter Triggered Queries Named as Command

In Spark 3.2.*x*, when DataFrameWriter triggered queries are sent to **QueryExecutionListener**, these queries are always named as **command**. In Spark 3.1 or earlier, these queries may be named as **save**, **insertInto**, or **saveAsTable**, depending on the specific operation.

- **Explanation:**

  When a query execution is triggered by DataFrameWriter, it is always named as **command** when sent to QueryExecutionListener.

  - **Spark 2.4.*x***: Named as **save**, **insertInto**, or **saveAsTable**.
  - **Spark 3.3.*x***: Named as **command**.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact.

Differences in Reading DATE and TIMESTAMP Fields

- **Explanation:**

  Differences in reading DATE and TIMESTAMP fields for the Asia/Shanghai time zone. For values before **1900-01-01 08:05:43**, values written by Spark 2.4.5 and read by Spark 3.3.1 differ from values read by Spark 2.4.5.

  - **Spark 2.4.*x***:

    For the Asia/Shanghai time zone, 1900-01-01 00:00:00, written in Spark 2.4.5 and read by Spark 2.4.5 returns **1900-01-01 00:00:00**.
  - **Spark 3.3.*x***:

    For the Asia/Shanghai time zone, 1900-01-01 00:00:00, written in Spark 2.4.5, read by Spark 3.3.1 with **spark.sql.parquet.int96RebaseModeInRead=LEGACY** returns **1900-01-01 00:00:00**. However, with **spark.sql.parquet.int96RebaseModeInRead=CORRECTED**, the resulting value is **1900-01-01 00:05:43**.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; the usage of DATE and TIMESTAMP fields needs to be evaluated.
- **Sample code**:

  Configure the following information in the job:

  ```
  spark.sql.session.timeZone=Asia/Shanghai
  ```

  - Spark 2.4.5

    ```
    spark.sql("create table parquet_timestamp_test (id int, col0 string, col1 timestamp) using
    parquet");
    spark.sql("insert into parquet_timestamp_test values (1, '245', '1900-01-01 00:00:00')");
    ```

    Execute SQL to read data:

```
spark.sql("select * from parquet_timestamp_test").show();
```

Query results:

```
+---+----+-------------------+
| id|col0|               col1|
+---+----+-------------------+
|  1| 245|1900-01-01 00:00:00|
+---+----+-------------------+
```

– Spark 3.3.1

```
spark.sql.parquet.int96RebaseModeInRead=LEGACY
```

Execute a job to read data:

```
spark.sql("select * from parquet_timestamp_test").show();
```

Query results

```
+---+----+-------------------+
| id|col0|               col1|
+---+----+-------------------+
|  1| 245|1900-01-01 00:00:00|
+---+----+-------------------+
```

Modify the configuration:

```
spark.sql.parquet.int96RebaseModeInRead=CORRECTED
```

Re-run the following SQL statement to read data:

```
spark.sql("select * from parquet_timestamp_test").show();
```

Query results:

```
+---+----+-------------------+
| id|col0|               col1|
+---+----+-------------------+
|  1| 245|1900-01-01 00:05:43|
+---+----+-------------------+
```

- **Configuration description:**

  These configuration items specify how Spark handles **DATE** and **TIMESTAMP** type fields for specific times (times that are contentious between the proleptic Gregorian and Julian calendars). For example, in the Asia/Shanghai time zone, it refers to how values before 1900-01-01 08:05:43 are processed.

  – **Configuration items in a datasource Parquet table**

Table 2-6 Configuration items in a Spark 3.3.1 datasource Parquet table

Configuration Item	Default Value	Description
spark.sql.parquet.int96RebaseModeInRead	**EXCEPTION** (default for Spark jobs)	Takes effect when reading INT96 type TIMESTAMP fields in Parquet files. ● **EXCEPTION**: Throws an error for specific times, causing the read operation to fail. ● **CORRECTED**: Reads the date/timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/timestamp from the traditional hybrid calendar (Julian + Gregorian) to the proleptic Gregorian calendar. This setting only takes effect when the write information for the Parquet file (e.g., Spark, Hive) is unknown.

Configuration Item	Default Value	Description
spark.sql.parquet.int96RebaseModeInWrite	**EXCEPTION** (default for Spark jobs)	Takes effect when writing INT96 type TIMESTAMP fields in Parquet files. ● **EXCEPTION**: Throws an error for specific times, causing the write operation to fail. ● **CORRECTED**: Writes the date/ timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/ timestamp from the proleptic Gregorian calendar to the traditional hybrid calendar (Julian + Gregorian) when writing Parquet files.

Configuration Item	Default Value	Description
spark.sql.parquet.datetimeRebaseModeIn-Read	**EXCEPTION** (default for Spark jobs)	Takes effect when reading **DATE**, **TIMESTAMP_MILLIS**, **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the read operation to fail. ● **CORRECTED**: Reads the date/ timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/ timestamp from the traditional hybrid calendar (Julian + Gregorian) to the proleptic Gregorian calendar. This setting only takes effect when the write information for the Parquet file (e.g., Spark, Hive) is unknown.

Configuration Item	Default Value	Description
spark.sql.parquet.datetimeRebaseModeIn-Write	**EXCEPTION** (default for Spark jobs)	Takes effect when writing **DATE**, **TIMESTAMP_MILLIS**, and **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the write operation to fail. ● **CORRECTED**: Writes the date/timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/timestamp from the proleptic Gregorian calendar to the traditional hybrid calendar (Julian + Gregorian) when writing Parquet files.

– **Configuration items in a datasource Avro table**

Table 2-7 Configuration items in a Spark 3.3.1 datasource Avro table

Configuration Item	Default Value	Description
spark.sql.avro.datetimeRebaseModeInRead	**EXCEPTION** (default for Spark jobs)	Takes effect when reading **DATE**, **TIMESTAMP_MILLIS**, **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the read operation to fail. ● **CORRECTED**: Reads the date/ timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/ timestamp from the traditional hybrid calendar (Julian + Gregorian) to the proleptic Gregorian calendar. This setting only takes effect when the write information for the Avro file (e.g., Spark, Hive) is unknown.

Configuration Item	Default Value	Description
spark.sql.avro.datetimeRebaseModeInWrite	**EXCEPTION** (default for Spark jobs)	Takes effect when writing **DATE**, **TIMESTAMP_MILLIS**, and **TIMESTAMP_MICROS** logical type fields. ● **EXCEPTION**: Throws an error for specific times, causing the write operation to fail. ● **CORRECTED**: Writes the date/timestamp as is without adjustment. ● **LEGACY**: Adjusts the date/timestamp from the proleptic Gregorian calendar to the traditional hybrid calendar (Julian + Gregorian) when writing Avro files.

Difference in from_unixtime Function

- **Explanation:**
  - **Spark 2.4.*x***:

    For the Asia/Shanghai time zone, **-2209017600** returns **1900-01-01 00:00:00**.
  - **Spark 3.3.*x***:

    For the Asia/Shanghai time zone, **-2209017943** returns **1900-01-01 00:00:00**.
- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; usage of this function needs to be checked.
- **Sample code**:

  Configure the following information in the job:

  ```
  spark.sql.session.timeZone=Asia/Shanghai
  ```
  - Spark 2.4.5

    Run the following statement to read data:

    ```
    select from_unixtime(-2209017600);
    ```
    Query results:

```
+-----------------------------------------------+
|from_unixtime(-2209017600, yyyy-MM-dd HH:mm:ss)|
+-----------------------------------------------+
|                            1900-01-01 00:00:00|
+-----------------------------------------------+
```

– Spark 3.3.1

  Run the following statement to read data:

```
select from_unixtime(-2209017600);
```

  Query results

```
+-----------------------------------------------+
|from_unixtime(-2209017600, yyyy-MM-dd HH:mm:ss)|
+-----------------------------------------------+
|                            1900-01-01 00:05:43|
+-----------------------------------------------+
```

Difference in unix_timestamp Function

- **Explanation:**

  For values less than **1900-01-01 08:05:43** in the Asia/Shanghai time zone.

  – **Spark 2.4.*x***:

    For the Asia/Shanghai time zone, **1900-01-01 00:00:00** returns **-2209017600**.

  – **Spark 3.3.*x***:

    For the Asia/Shanghai time zone, **1900-01-01 00:00:00** returns **-2209017943**.

- **Is there any impact on jobs after the engine version upgrade?**

  There is an impact; usage of this function needs to be checked.

- **Sample code**:

  Configure the following information in the job:

```
spark.sql.session.timeZone=Asia/Shanghai
```

  – Spark 2.4.5

    Run the following statement to read data:

```
select unix_timestamp('1900-01-01 00:00:00');
```

    Query results:

```
+-------------------------------------------------------+
|unix_timestamp(1900-01-01 00:00:00, yyyy-MM-dd HH:mm:ss)|
+-------------------------------------------------------+
|                                            -2209017600|
+-------------------------------------------------------+
```

  – Spark 3.3.1

    Run the following statement to read data:

```
select unix_timestamp('1900-01-01 00:00:00');
```

    Query results

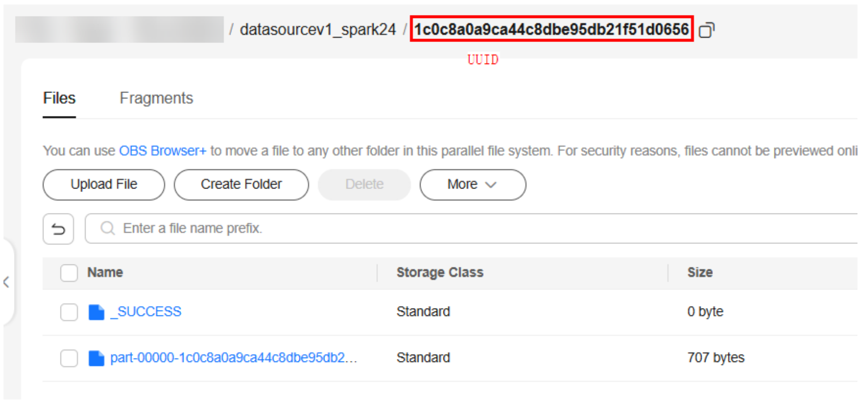
```
+-------------------------------------------------------+
|unix_timestamp(1900-01-01 00:00:00, yyyy-MM-dd HH:mm:ss)|
+-------------------------------------------------------+
|                                            -2209017943|
+-------------------------------------------------------+
```

2.7.3 DLI Datasource V1 Table and Datasource V2 Table

What Are DLI Datasource V1 and V2 Tables?

- DLI datasource V1 table (referred to as V1 table): This is a DLI-specific datasource table format. DLI's custom create/insert/truncate commands are used, and the data path of the table is **$tablepath/UUID/***Data file*.

Figure 2-2 DLI datasource v1 table



[redacted] / datasourcev1_spark24 / 1c0c8a0a9ca44c8dbe95db21f51d0656

UUID

Files Fragments

You can use OBS Browser+ to move a file to any other folder in this parallel file system. For security reasons, files cannot be previewed onli

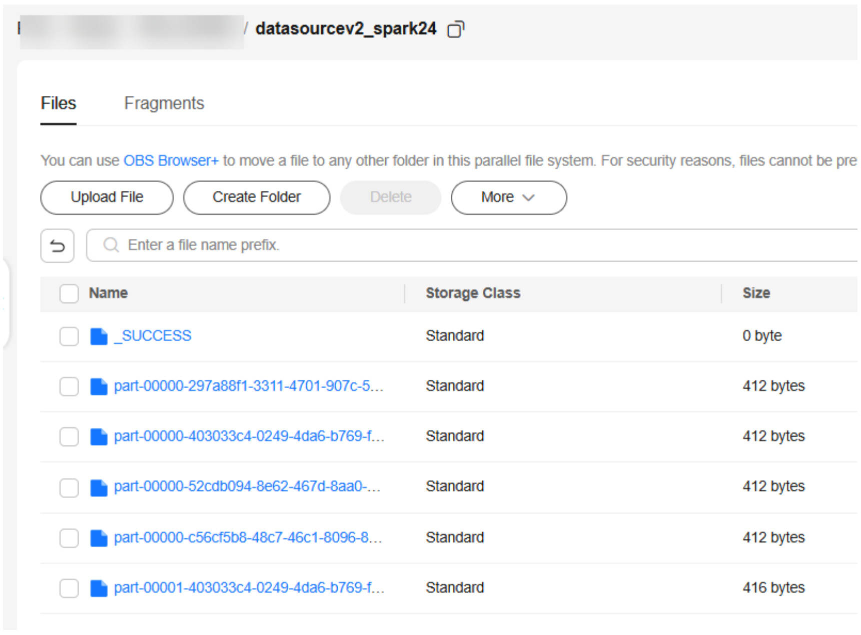
Upload File | Create Folder | Delete | More

Enter a file name prefix.

Name	Storage Class	Size
_SUCCESS	Standard	0 byte
part-00000-1c0c8a0a9ca44c8dbe95db2...	Standard	707 bytes

- DLI datasource V2 table (referred to as V2 table): This is the open-source datasource table format of Spark. Spark's open-source create/insert/truncate commands are used, and the data path of the table is **$tablepath/***Data file*.

Figure 2-3 DLI datasource v2 table



[redacted] / datasourcev2_spark24

Files Fragments

You can use OBS Browser+ to move a file to any other folder in this parallel file system. For security reasons, files cannot be pre

Upload File | Create Folder | Delete | More

Enter a file name prefix.

Name	Storage Class	Size
_SUCCESS	Standard	0 byte
part-00000-297a88f1-3311-4701-907c-5...	Standard	412 bytes
part-00000-403033c4-0249-4da6-b769-f...	Standard	412 bytes
part-00000-52cdb094-8e62-467d-8aa0-...	Standard	412 bytes
part-00000-c56cf5b8-48c7-46c1-8096-8...	Standard	412 bytes
part-00001-403033c4-0249-4da6-b769-f...	Standard	416 bytes

Compatibility of DLI Spark Versions with V1 and V2 Tables

Table 2-8 Compatibility of DLI Spark versions with v1 and v2 tables

Table Type	Spark 2.3 SQL Queue	Spark 2.3 General-Purpose Queue	Spark 2.4 SQL Queue	Spark 2.4 General-Purpose Queue	Spark 3.1 SQL Queue	Spark 3.1 General-Purpose Queue	Spark 3.3 SQL Queue	Spark 3.3 General-Purpose Queue
V1 table	√	√	√	√	√	√	√	Partially supported
V2 table	×	×	√	√	×	×	√	√

Table 2-9 Syntax support list for Spark 3.3 general-purpose queues

Table Type	select	create table	create table like	CTAS	insert into	insert overwrite	load data	alter table set location	truncate table
V1 table	√	√	√	×	×	×	×	×	×
V2 table	√	√	√	√	√	√	√	√	√

How Do I Confirm If a User-Created Table is a V1 or V2 Table?

1. Use the datasource syntax to create a table:

```
CREATE TABLE IF NOT EXISTS table_name (id STRING) USING parquet;
```

2. Run **show create table** to check the value of the **version** field under **TBLPROPERTIES**.

If **v1**, it is a V1 table; if **v2**, it is a V2 table.

To change a V1 table to a V2 table, submit a service ticket to contact customer support.

Example Upgrade

NOTE

Upgrading the Spark engine and modifying data tables may cause changes in the cost of billed resources if the type of compute resource changes when creating a queue.

- If the original queue uses compute resources of the elastic resource pool type, creating a queue does not involve changes in the cost of compute resources.
- If the original queue uses compute resources of a non-elastic resource pool type, creating a queue within an elastic resource pool will change the cost of compute resources. Refer to the price details of compute resources for specifics.

- **Example 1: Does upgrading Spark from version 2.4.x to Spark 3.3.1 affect the version of data tables when using a SQL queue?**

  No, SQL queues in Spark 2.4.*x* support V1 and V2 tables, so upgrading Spark only requires considering the compatibility of the Spark version with SQL syntax.

- **Example 2: Does upgrading Spark from version 2.4.x to Spark 3.3.1 affect the version of data tables when using a general-purpose queue?**

  General-purpose queues in Spark 2.4.*x* support V1 and V2 tables, but general-purpose queues in Spark 3.3.*x* do not support V1 tables.

  Therefore, to upgrade Spark from version 2.4.*x* to 3.3.1, follow these steps:

  a. Change V1 tables in Spark 2.4.*x* to V2 tables.
  b. Upgrade V2 tables in Spark 2.4.*x* to V2 tables in Spark 3.3.1.

     Consider the compatibility of Spark Jar job API syntax as well.

  Table 2-10 Compatibility of DLI Spark versions with v1 and v2 tables

Table Type	Spark 2.4 General-Purpose Queue	Spark 3.3 General-Purpose Queue
V1 table	√	Partially supported
V2 table	√	√

- **Example 3: How do I upgrade V1 tables in Spark 2.3.2 to V2 tables in Spark 3.3.1 using a general-purpose queue?**

  General-purpose queues in Spark 2.3.2 do not support V2 tables, and general-purpose queues in Spark 3.3.1 do not support V1 tables.

  a. Upgrade V1 tables in Spark 2.3.2 to V1 tables in Spark 2.4.5.
  b. Change V1 tables in Spark 2.4.5 to V2 tables.
  c. Upgrade V2 tables in Spark 2.4.5 to V2 tables in Spark 3.3.1.

     Consider the compatibility of Spark Jar job API syntax as well.

Table 2-11 Compatibility of DLI Spark versions with v1 and v2 tables

Table Type	Spark 2.3 General-Purpose Queue	Spark 2.4 General-Purpose Queue	Spark 3.3 General-Purpose Queue
V1 table	√	√	Partially supported
V2 table	×	√	√