



Using Apache Iceberg on AWS

AWS Prescriptive Guidance



AWS Prescriptive Guidance: Using Apache Iceberg on AWS

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Introduction	1
Modern data lakes	3
Advanced use cases in modern data lakes	3
Introduction to Apache Iceberg	4
AWS support for Apache Iceberg	5
Getting started with Iceberg tables in Athena SQL	8
Creating an unpartitioned table	8
Creating a partitioned table	9
Creating a table and loading data with a single CTAS statement	9
Inserting, updating, and deleting data	10
Querying Iceberg tables	11
Iceberg table anatomy	11
Working with Iceberg in Amazon EMR	14
Version and feature compatibility	14
Creating an Amazon EMR cluster with Iceberg	14
Developing Iceberg applications in Amazon EMR	15
Using Amazon EMR Studio notebooks	15
Running Iceberg jobs in Amazon EMR	16
Best practices for Amazon EMR	20
Working with Iceberg in AWS Glue	22
Using native Iceberg integration	22
Using a custom Iceberg version	23
Spark configurations for Iceberg in AWS Glue	24
Best practices for AWS Glue jobs	25
Working with Iceberg tables by using Spark	27
Creating and writing Iceberg tables	27
Using Spark SQL	27
Using the DataFrames API	28
Updating data in Iceberg tables	29
Upserting data in Iceberg tables	30
Deleting data in Iceberg tables	30
Reading data	31
Using time travel	31
Using incremental queries	32

Accessing metadata	33
Working with Iceberg tables by using Trino	34
Amazon EMR on EC2 setup	34
Creating Iceberg tables	35
Reading from Iceberg tables	36
Upserting data into Iceberg tables	36
Deleting records from Iceberg tables	37
Querying Iceberg table metadata	37
Using time travel	38
Considerations when using Iceberg with Trino	38
Working with Iceberg tables by using Firehose	39
Working with Iceberg tables by using Athena SQL	40
Version and feature compatibility	40
Iceberg table specification support	40
Iceberg feature support	40
Working with Iceberg tables	41
Working with Iceberg tables by using Pylceberg	43
Prerequisites	43
Connecting to the Data Catalog	43
Listing and creating databases	44
Creating and writing Iceberg tables	44
Unpartitioned tables	44
Partitioned tables	45
Reading data	48
Deleting data	48
Accessing metadata	48
Using time travel	49
Working with Iceberg table format specification version 3	50
Key features in version 3	50
Version compatibility	51
Getting started with version 3	51
Prerequisites	51
Creating version 3 tables	51
Enabling deletion vectors	53
Using row lineage for change tracking	53
Best practices for version 3	54

When to use version 3	54
Optimizing write performance	54
Optimizing read performance	54
Migration strategy	55
Compatibility considerations	55
Troubleshooting	55
Common issues	55
Getting help	56
Pricing	56
Availability	57
Additional resources	57
Migrating existing tables to Iceberg	58
In-place migration	58
Option 1: snapshot procedure	60
Option 2: migrate procedure	62
Replicating the table migration procedure in AWS Glue Data Catalog	65
Keeping Iceberg tables in sync after in-place migration	66
Choosing the right in-place migration strategy	68
Full data migration	70
Choosing a migration strategy	71
Migration options summary	72
Best practices for optimizing Iceberg workloads	78
General best practices	78
Optimizing read performance	79
Partitioning	79
Tuning file sizes	81
Optimize column statistics	83
Choose the right update strategy	84
Use ZSTD compression	84
Set the sort order	84
Optimizing write performance	87
Set the table distribution mode	87
Choose the right update strategy	87
Choose the right file format	88
Optimizing storage	89
Enable S3 Intelligent-Tiering	89

Archive or delete historic snapshots	89
Delete orphan files	93
Maintaining tables by using compaction	93
Iceberg compaction	94
Tuning compaction behavior	95
Running compaction with Spark on Amazon EMR or AWS Glue	97
Running compaction with Amazon Athena	97
Recommendations for running compaction	98
Using Iceberg workloads in Amazon S3	99
Prevent hot partitioning (HTTP 503 errors)	99
Use Iceberg maintenance operations to release unused data	100
Replicate data across AWS Regions	100
Monitoring Iceberg workloads	102
Table-level monitoring	102
Database-level monitoring	104
Preventive maintenance	105
Governance and access control	107
Reference architectures	108
Nightly batch ingestion	108
Data lake that combines batch and near real-time ingestion	109
Resources	110
Contributors	111
Document history	113

Using Apache Iceberg on AWS

Amazon Web Services ([contributors](#))

November 2025 ([document history](#))

Apache Iceberg is an open-source table format that simplifies table management while improving performance. AWS analytics services such as Amazon EMR, AWS Glue, Amazon Athena, and Amazon Redshift include native support for Iceberg, so you can easily build transactional data lakes on top of Amazon Simple Storage Service (Amazon S3) on AWS.

In addition, the next generation of Amazon SageMaker is built on an [open lakehouse architecture](#) that unifies data access across AWS data lakes, data warehouses, and third-party and federated sources. The lakehouse is fully compatible with Iceberg and gives you the flexibility to access and query data in place by using the Iceberg REST API.

This technical guide provides guidance on getting started with Iceberg on different AWS services, and includes best practices and recommendations for running Iceberg on AWS at scale while optimizing cost and performance.

Whether you're just starting out with Iceberg or you're an experienced user looking to optimize your existing Iceberg workloads on AWS, this guide offers valuable insights for every stage of your project

In this guide:

- [Modern data lakes](#)
- [Getting started with Iceberg tables in Athena SQL](#)
- [Working with Iceberg in Amazon EMR](#)
- [Working with Iceberg in AWS Glue](#)
- [Working with Iceberg tables by using Spark](#)
- [Working with Iceberg tables by using Trino](#)
- [Working with Iceberg tables by using Amazon Data Firehose](#)
- [Working with Iceberg tables by using Athena SQL](#)
- [Working with Iceberg tables by using Pylceberg](#)
- [Working with Iceberg table format specification version 3](#)

- [Migrating existing tables to Iceberg](#)
- [Best practices for optimizing Iceberg workloads](#)
- [Monitoring Iceberg workloads](#)
- [Governance and access control](#)
- [Reference architectures](#)
- [Resources](#)
- [Contributors](#)

Modern data lakes

Advanced use cases in modern data lakes

The evolution of data storage has progressed from databases to data warehouses and data lakes, where each technology addresses unique business and data requirements. Traditional databases excelled at handling structured data and transactional workloads, but they faced performance challenges as data volumes increased. Data warehouses emerged to tackle performance and scalability issues, but like databases, they relied on proprietary formats within vertically integrated systems.

Data lakes offer one of the best options for storing data in terms of cost, scalability, and flexibility. You can use a data lake to retain large volumes of structured and unstructured data at a low cost, and use this data for different types of analytics workloads, from business intelligence reporting to big data processing, real-time analytics, machine learning, and generative artificial intelligence (AI), to help guide better decisions.

Despite these benefits, data lakes weren't initially designed with database-like capabilities. A data lake doesn't provide support for atomicity, consistency, isolation, and durability (ACID) processing semantics, which you might require to optimize and manage your data effectively at scale across hundreds or thousands of users by using many different technologies. Data lakes don't provide native support for the following functionality:

- Performing efficient record-level updates and deletions as data changes in your business
- Managing query performance as tables grow to millions of files and hundreds of thousands of partitions
- Ensuring data consistency across multiple concurrent writers and readers
- Preventing data corruption when write operations fail partway through the operation
- Evolving table schemas over time without (partially) rewriting datasets

These challenges have become particularly prevalent in use cases such as handling change data capture (CDC) or use cases pertaining to privacy, deletion of data, and streaming data ingestion, which can result in sub-optimal tables.

Data lakes that use the traditional Hive-format tables support write operations only for entire files. This makes updates and deletes difficult to implement, time consuming, and costly. Moreover,

concurrency controls and guarantees offered in ACID-compliant systems are needed to ensure data integrity and consistency.

These challenges leave users with a dilemma: choose between a fully integrated but proprietary platform, or opt for a vendor-neutral but resource-intensive, self-built data lake that requires constant maintenance and migration to realize its potential value.

To help overcome these challenges, Iceberg provides additional database-like functionality that simplifies the optimization and management overhead of data lakes, while still supporting storage on cost-effective systems such as [Amazon S3](#).

Introduction to Apache Iceberg

Apache Iceberg is an open-source table format that provides features in data lake tables that were historically only available in databases or data warehouses. It's designed for scale and performance, and is well-suited for managing tables that are over hundreds of gigabytes. Some of the main features of Iceberg tables are:

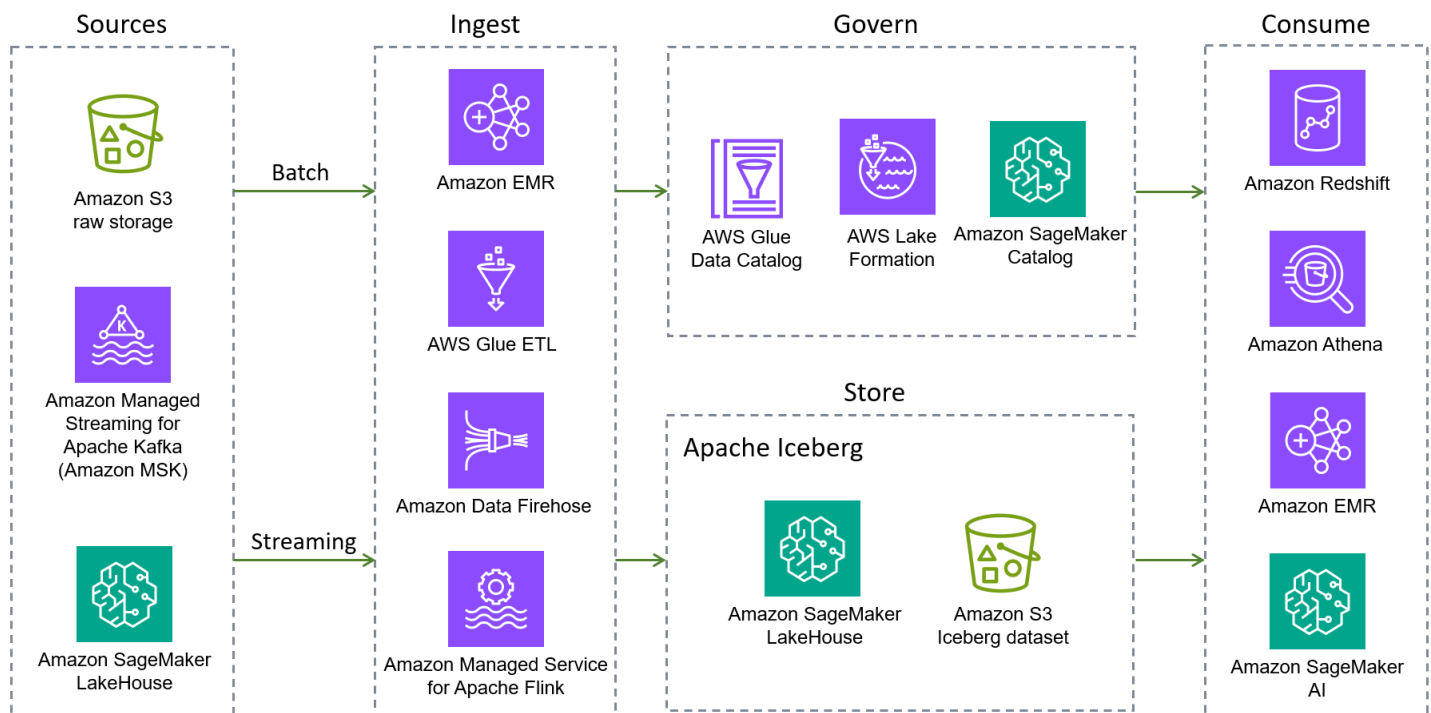
- **Delete, update, and merge.** Iceberg supports standard SQL commands for data warehousing for use with data lake tables.
- **Fast scan planning and advanced filtering.** Iceberg stores metadata such as partition and column-level statistics that can be used by engines to speed up planning and running queries.
- **Full schema evolution.** Iceberg supports adding, dropping, updating, or renaming columns without side-effects.
- **Partition evolution.** You can update the partition layout of a table as data volume or query patterns change. Iceberg supports changing the columns that a table is partitioned on, or adding columns to, or removing columns from, composite partitions.
- **Hidden partitioning.** This feature prevents reading unnecessary partitions automatically. This eliminates the need for users to understand the table's partitioning details or to add extra filters to their queries.
- **Version rollback.** Users can quickly correct problems by reverting to a pre-transaction state.
- **Time travel.** Users can query a specific previous version of a table.
- **Serializable isolation.** Table changes are atomic, so readers never see partial or uncommitted changes.
- **Concurrent writers.** Iceberg uses optimistic concurrency to allow multiple transactions to succeed. In case of conflicts, one of the writers has to retry the transaction.

- **Open file formats.** Iceberg supports multiple open source file formats, including [Apache Parquet](#), [Apache Avro](#), and [Apache ORC](#).

In summary, data lakes that use the Iceberg format benefit from transactional consistency, speed, scale, and schema evolution. For more information about these and other Iceberg features, see the [Apache Iceberg documentation](#).

AWS support for Apache Iceberg

Apache Iceberg is supported by AWS services such as [Amazon EMR](#), [Amazon Athena](#), [Amazon Redshift](#), [AWS Glue](#), and [Amazon SageMaker](#). The following diagram depicts a simplified reference architecture of a data lake that's based on Iceberg.



The following AWS services provide native Iceberg integrations. There are additional AWS services that can interact with Iceberg, either indirectly or by packaging the Iceberg libraries.

- [Amazon S3](#) is the best place to build data lakes because of its durability, availability, scalability, security, compliance, and audit capabilities. Iceberg was designed and built to interact with Amazon S3 seamlessly, and provides support for many Amazon S3 features as listed in the [Iceberg documentation](#). In addition, [Amazon S3 Tables](#) deliver the first cloud object store with built-in Iceberg support and streamline storing tabular data at scale. With S3 Tables support

for Iceberg, you can easily query your tabular data by using popular AWS and third-party query engines.

- [The next generation of SageMaker](#) is built on an open lakehouse architecture that unifies data access across Amazon S3 data lakes, Amazon Redshift data warehouses, and third-party and federated data sources. These capabilities help you build powerful analytics and AI/ML applications on a single copy of data. The lakehouse is fully compatible with Iceberg, so you have the flexibility to access and query data in place by using the Iceberg REST API.
- [Amazon EMR](#) is a big data solution for petabyte-scale data processing, interactive analytics, and machine learning by using open source frameworks such as Apache Spark, Flink, Trino, and Hive. Amazon EMR can run on customized Amazon Elastic Compute Cloud (Amazon EC2) clusters, Amazon Elastic Kubernetes Service (Amazon EKS), AWS Outposts, or Amazon EMR Serverless.
- [Amazon Athena](#) is a serverless, interactive analytics service that's built on open source frameworks. It supports open-table and file formats and provides a simplified, flexible way to analyze petabytes of data where it lives. Athena provides native support for read, time travel, write, and DDL queries for Iceberg and uses the AWS Glue Data Catalog for the Iceberg metastore.
- [Amazon Redshift](#) is a petabyte-scale cloud data warehouse that supports both cluster-based and serverless deployment options. Amazon Redshift Spectrum can query external tables that are registered with the AWS Glue Data Catalog and stored on Amazon S3. Redshift Spectrum also provides support for the Iceberg storage format.
- [AWS Glue](#) is a serverless data integration service that makes it easier to discover, prepare, move, and integrate data from multiple sources for analytics, machine learning (ML), and application development. It is fully integrated with Iceberg. Specifically, you can perform read and write operations on Iceberg tables by using AWS Glue jobs, manage tables through the [AWS Glue Data Catalog](#) (Hive metastore-compatible), discover and register tables automatically by using AWS Glue crawlers, and evaluate data quality in Iceberg tables through the AWS Glue Data Quality feature. The AWS Glue Data Catalog also supports collecting column statistics, calculating and updating the number of distinct values (NDVs) for each column in Iceberg tables, and automatic table optimizations (compaction, snapshot retention, orphan file deletion). AWS Glue also supports zero-ETL integrations from a list of AWS services and third-party applications into Iceberg tables.
- [Amazon Data Firehose](#) is a fully managed service for delivering real-time streaming data to destinations such as Amazon S3, Amazon Redshift, Amazon OpenSearch Service, Amazon OpenSearch Serverless, Splunk, Apache Iceberg tables, and any custom HTTP or HTTPS endpoints owned by supported third-party service providers, including Datadog, Dynatrace, LogicMonitor,

MongoDB, New Relic, Coralogix, and Elastic. With Firehose, you don't need to write applications or manage resources. You configure your data producers to send data to Firehose, and it automatically delivers the data to the destination that you specified. You can also configure Firehose to transform your data before delivering it.

- [Amazon Managed Service for Apache Flink](#) is a fully managed Amazon service that lets you use an Apache Flink application to process streaming data. It supports both reading from and writing to Iceberg tables, and enables real-time data processing and analytics.
- [Amazon SageMaker AI](#) supports the storage of feature sets in Amazon SageMaker AI Feature Store by using Iceberg format.
- [AWS Lake Formation](#) provides coarse and fine-grained access control permissions to access data, including Iceberg tables consumed by Athena or Amazon Redshift. To learn more about permissions support for Iceberg tables, see the [Lake Formation documentation](#).

AWS has a wide range of services that support Iceberg, but covering all these services is beyond the scope of this guide. The following sections cover Spark (batch and structured streaming) on Amazon EMR and AWS Glue, as well as Athena SQL. The [following section](#) provides a quick look at Iceberg support in Athena SQL.

Getting started with Iceberg tables in Amazon Athena SQL

Amazon Athena provides built-in support for Iceberg. You can use Iceberg without any additional steps or configuration except for setting up the service prerequisites detailed in the [Getting started](#) section of the Athena documentation. This section provides a brief introduction to creating tables in Athena. For more information, see [Working with Iceberg tables by using Athena SQL](#) later in this guide.

You can create Iceberg tables on AWS by using different engines. Those tables work seamlessly across AWS services. To create your first Iceberg tables with Athena SQL, you can use the following boilerplate code.

```
CREATE TABLE <table_name> (  
    col_1 string,  
    col_2 string,  
    col_3 bigint,  
    col_ts timestamp)  
PARTITIONED BY (col_1, <<<partition_transform>>>(col_ts))  
LOCATION 's3://<bucket>/<folder>/<table_name>/'  
TBLPROPERTIES (  
    'table_type' = 'ICEBERG'  
)
```

The following sections provide examples of creating partitioned and unpartitioned Iceberg table in Athena. For more information, see the Iceberg syntax detailed in the [Athena documentation](#).

Creating an unpartitioned table

The following example statement customizes the boilerplate SQL code to create an unpartitioned Iceberg table in Athena. You can add this statement to the query editor in the [Athena console](#) to create the table.

```
CREATE TABLE athena_iceberg_table (  
    color string,  
    date string,  
    name string,  
    price bigint,
```

```
    product string,  
    ts timestamp)  
LOCATION 's3://DOC_EXAMPLE_BUCKET/ice_warehouse/iceberg_db/athena_iceberg_table/'  
TBLPROPERTIES (  
    'table_type' = 'ICEBERG'  
)
```

For step-by-step instructions for using the query editor, see [Getting started](#) in the Athena documentation.

Creating a partitioned table

The following statement creates a partitioned table based on the date by using Iceberg's concept of [hidden partitioning](#). It uses the `day()` transform to derive daily partitions, using the `dd-mm-yyyy` format, out of a timestamp column. Iceberg doesn't store this value as a new column in the dataset. Instead, the value is derived on the fly when you write or query data.

```
CREATE TABLE athena_iceberg_table_partitioned (  
    color string,  
    date string,  
    name string,  
    price bigint,  
    product string,  
    ts timestamp)  
PARTITIONED BY (day(ts))  
LOCATION 's3://DOC_EXAMPLE_BUCKET/ice_warehouse/iceberg_db/athena_iceberg_table/'  
TBLPROPERTIES (  
    'table_type' = 'ICEBERG'  
)
```

Creating a table and loading data with a single CTAS statement

In the partitioned and unpartitioned examples in the previous sections, the Iceberg tables are created as empty tables. You can load data to the tables by using the `INSERT` or `MERGE` statement. Alternatively, you can use a `CREATE TABLE AS SELECT` (CTAS) statement to create and load data into an Iceberg table in a single step.

CTAS is the best way in Athena to create a table and load data in a single statement. The following example illustrates how to use CTAS to create an Iceberg table (`iceberg_ctas_table`) from an existing Hive/Parquet table (`hive_table`) in Athena.

```
CREATE TABLE iceberg_ctas_table WITH (  
  table_type = 'ICEBERG',  
  is_external = false,  
  location = 's3://DOC_EXAMPLE_BUCKET/ice_warehouse/iceberg_db/iceberg_ctas_table/'  
) AS  
SELECT * FROM "iceberg_db"."hive_table" limit 20  
---  
SELECT * FROM "iceberg_db"."iceberg_ctas_table" limit 20
```

To learn more about CTAS, see the [Athena CTAS documentation](#).

Inserting, updating, and deleting data

Athena supports different ways of writing data to an Iceberg table by using the `INSERT INTO`, `UPDATE`, `MERGE INTO`, and `DELETE FROM` statements.

Note

Athena SQL doesn't currently support the copy-on-write approach. `UPDATE`, `MERGE INTO`, and `DELETE FROM` operations always use the merge-on-read approach with positional deletes, regardless of specified table properties. If you set up table properties such as `write.update.mode`, `write.merge.mode`, and `write.delete.mode` to use copy-on-write, your queries won't fail, but Athena will ignore them and keep using merge-on-read.

The following statement uses `INSERT INTO` to add data to an Iceberg table:

```
INSERT INTO "iceberg_db"."ice_table" VALUES (  
  'red', '222022-07-19T03:47:29', 'PersonNew', 178, 'Tuna', now()  
)  
  
SELECT * FROM "iceberg_db"."ice_table"  
where color = 'red' limit 10;
```

Sample output:

#	color	date	name	price	product	ts
1	red	222022-07-19T03:47:29	PersonNew	178	Tuna	2023-10-11 11:35:01.298000 UTC

For more information, see the [Athena documentation](#).

Querying Iceberg tables

You can run regular SQL queries against your Iceberg tables by using Athena SQL, as illustrated in the previous example.

In addition to the usual queries, Athena also supports time travel queries for Iceberg tables. As discussed previously, you can change existing records through updates or deletes in an Iceberg table, so it's convenient to use time travel queries to look back into older versions of your table based on a timestamp or a snapshot ID.

For example, the following statement updates a color value for Person5, and then displays an earlier value from January 4, 2023:

```
UPDATE ice_table SET color='new_color' WHERE name='Person5'  
  
SELECT * FROM "iceberg_db"."ice_table" FOR TIMESTAMP AS OF TIMESTAMP '2023-01-04  
12:00:00 UTC'
```

Sample output:

Results (15)								Copy	Download results
#	color	date	name	price	product	ts			
1	cyan	222022-07-19T03:47:29	Person5	353	Keyboard	2023-01-03 10:15:52.268000 UTC			
2	lime	222022-07-19T03:47:29	Person1	833	Towels	2023-01-03 10:15:52.268000 UTC			
3	turquoise	222022-07-19T03:47:29	Person1	1319	Shirt	2023-01-03 10:15:52.268000 UTC			
4	blue	222022-07-19T03:47:29	Person3	163	Sausages	2023-01-03 10:15:52.268000 UTC			

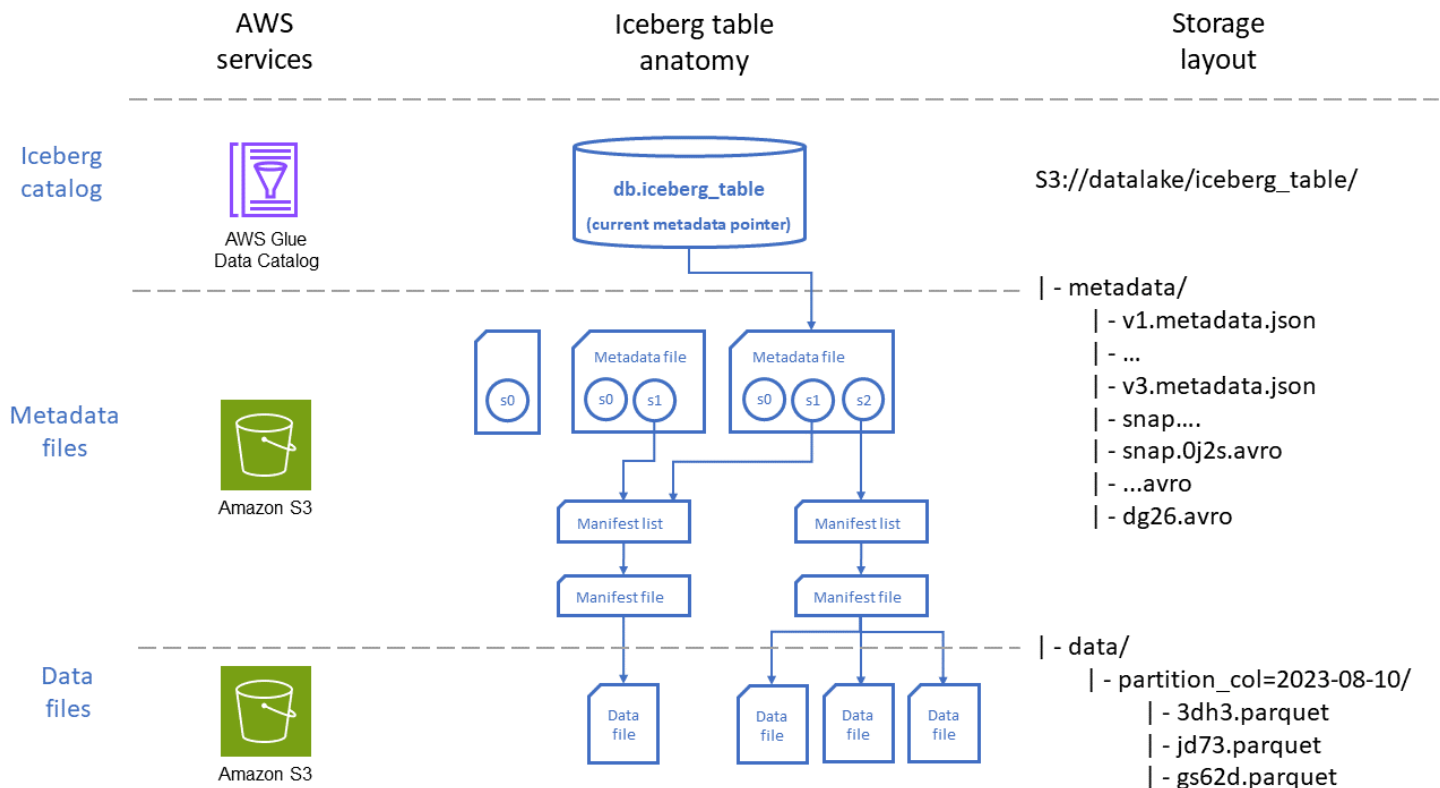
For syntax and additional examples of time travel queries, see the [Athena documentation](#).

Iceberg table anatomy

Now that we've covered the basic steps of working with Iceberg tables, let's dive deeper into the intricate details and design of an Iceberg table.

To enable the features [described earlier](#) in this guide, Iceberg is designed with hierarchical layers of data and metadata files. These layers manage metadata intelligently to optimize query planning and execution.

The following diagram portrays the organization of an Iceberg table through two perspectives: the AWS services used to store the table and the file placement in Amazon S3.



As shown in the diagram, an Iceberg table consists of three main layers:

- **Iceberg catalog:** AWS Glue Data Catalog integrates natively with Iceberg and is, for most use cases, the best option for workloads that run on AWS. Services that interact with Iceberg tables (for example, Athena) use the catalog to find the current snapshot version of the table, either to read or to write data.
- **Metadata layer:** Metadata files, namely the manifest files and manifest list files, keep track of information such as the schema of the tables, the partition strategy, and the location of the data files, as well as column-level statistics such as minimum and maximum ranges for the records that are stored in each data file. These metadata files are stored in Amazon S3 within the table path.
 - **Manifest files** contain a record for each data file, including its location, format, size, checksum, and other relevant information.
 - **Manifest lists** provide an index of the manifest files. As the number of manifest files grows in a table, breaking up that information into smaller subsections helps reduce the number of manifest files that need to be scanned by queries.

- **Metadata files** contain information about the whole Iceberg table, including the manifest lists, the schemas, partition metadata, snapshot files, and other files that are used to manage the table's metadata.
- **Data layer:** This layer contains the files that have the data records that queries will run against. These files can be stored in different formats, including [Apache Parquet](#), [Apache Avro](#), and [Apache ORC](#).
- **Data files** contain the data records for a table.
- **Delete files** encode row-level delete and update operations in an Iceberg table. Iceberg has two types of delete files, as described in the [Iceberg documentation](#). These files are created by operations by using the merge-on-read mode.

Working with Iceberg in Amazon EMR

Amazon EMR provides petabyte-scale data processing, interactive analytics, and machine learning in the cloud by using open source frameworks such as Apache Spark, Apache Hive, Flink, and Trino.

Note

This guide uses Apache Spark for examples.

Amazon EMR supports multiple deployment options: Amazon EMR on EC2, Amazon EMR on EKS, Amazon EMR Serverless, and Amazon EMR on AWS Outposts. To choose a deployment option for your workload, see the [Amazon EMR FAQ](#).

Version and feature compatibility

Amazon EMR version 6.5.0 and later versions support Apache Iceberg natively. For a list of supported Iceberg versions for each Amazon EMR release, see [Iceberg release history](#) in the Amazon EMR documentation. Also review the sections under [Use a cluster with Iceberg](#) to see which Iceberg features are supported in Amazon EMR on different frameworks.

We recommend that you use the latest Amazon EMR version to benefit from the latest supported Iceberg version. The code examples and configurations in this section assume that you're using Amazon EMR release **emr-7.8.0**.

Creating an Amazon EMR cluster with Iceberg

To create an Amazon EMR cluster on Amazon EC2 with Iceberg installed, follow the instructions in the [Amazon EMR documentation](#).

Specifically, your cluster should be configured with the following classification:

```
[{
  "Classification": "iceberg-defaults",
  "Properties": {
    "iceberg.enabled": "true"
  }
}]
```

You can also choose to use Amazon EMR Serverless or Amazon EMR on EKS as deployment options for your Iceberg workloads, starting from Amazon EMR 6.6.0.

Developing Iceberg applications in Amazon EMR

To develop the Spark code for your Iceberg applications, you can use [Amazon EMR Studio](#), which is a web-based integrated development environment (IDE) for fully managed Jupyter notebooks that run on Amazon EMR clusters.

Using Amazon EMR Studio notebooks

You can interactively develop Spark applications in Amazon EMR Studio Workspace notebooks and connect those notebooks to your Amazon EMR on EC2 clusters or Amazon EMR on EKS managed endpoints. See AWS service documentation for instructions on setting up an EMR Studio for [Amazon EMR on EC2](#) and [Amazon EMR on EKS](#).

To use Iceberg in EMR Studio, follow these steps:

1. Launch an Amazon EMR cluster with Iceberg enabled, as instructed in [Use a cluster with Iceberg Installed](#).
2. Set up an EMR Studio. For instructions, see [Set up an Amazon EMR Studio](#).
3. Open an EMR Studio Workspace notebook and run the following code as the first cell in the notebook to configure your Spark session for using Iceberg:

```
%%configure -f
{
  "conf": {
    "spark.sql.catalog.<catalog_name>": "org.apache.iceberg.spark.SparkCatalog",
    "spark.sql.catalog.<catalog_name>.warehouse": "s3://YOUR-BUCKET-NAME/YOUR-
FOLDER-NAME/",
    "spark.sql.catalog.<catalog_name>.type": "glue",
    "spark.sql.extensions":
    "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
  }
}
```

where:

- `<catalog_name>` is your Iceberg Spark session catalog name. Replace it with a name of your choice, and remember to change the references throughout all configurations that are

associated with this catalog. In your code, you can refer to your Iceberg tables by using the fully qualified table name, including the Spark session catalog name, as follows:

```
<catalog_name>.<database_name>.<table_name>
```

Alternatively, you can change the default catalog to the Iceberg catalog that you defined by setting `spark.sql.defaultCatalog` to your catalog name. This second approach enables you to refer to tables without the catalog prefix, which can simplify your queries.

- `<catalog_name>.warehouse` points to the Amazon S3 path where you want to store your data and metadata.
 - To make the catalog an AWS Glue Data Catalog, set `spark.sql.catalog.<catalog_name>.type` to `glue`. This key is required to point to an implementation class for any custom catalog implementation. The [General best practices](#) section later in this guide describes the different Iceberg-supported catalogs.
4. You can now start interactively developing your Spark application for Iceberg in the notebook, as you would for any other Spark application.

For more information about configuring Spark for Apache Iceberg by using Amazon EMR Studio, see the blog post [Build a high-performance, ACID compliant, evolving data lake using Apache Iceberg on Amazon EMR](#).

Running Iceberg jobs in Amazon EMR

After you develop the Spark application code for your Iceberg workload, you can run it on any Amazon EMR deployment option that supports Iceberg (see the [Amazon EMR FAQ](#)).

As with other Spark jobs, you can submit work to an Amazon EMR on EC2 cluster by adding steps or by interactively submitting Spark jobs to the master node. To run a Spark job, see the following Amazon EMR documentation pages:

- For an overview of the different options for submitting work to an Amazon EMR on EC2 cluster and detailed instructions for each option, see [Submit work to a cluster](#).
- For Amazon EMR on EKS, see [Running Spark jobs with StartJobRun](#).
- For EMR Serverless, see [Running jobs](#).

The following sections provide an example for each Amazon EMR deployment option.

Amazon EMR on EC2

You can use these steps to submit the Iceberg Spark job:

1. Create the file `emr_step_iceberg.json` with the following content on your workstation:

```
[{
  "Name": "iceberg-test-job",
  "Type": "spark",
  "ActionOnFailure": "CONTINUE",
  "Args": [
    "--deploy-mode",
    "client",
    "--conf",

    "spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions",
    "--conf",
    "spark.sql.catalog.<catalog_name>=org.apache.iceberg.spark.SparkCatalog",
    "--conf",
    "spark.sql.catalog.<catalog_name>.type=glue",
    "--conf",
    "spark.sql.catalog.<catalog_name>.warehouse=s3://YOUR-BUCKET-NAME/YOUR-
FOLDER-NAME/",
    "s3://YOUR-BUCKET-NAME/code/iceberg-job.py"
  ]
}]
```

2. Modify the configuration file for your specific Spark job by customizing the Iceberg configuration options that are highlighted in bold.
3. Submit the step by using the AWS Command Line Interface (AWS CLI). Run the command in the directory where the `emr_step_iceberg.json` file is located.

```
aws emr add-steps --cluster-id <cluster_id> --steps file://emr_step_iceberg.json
```

Amazon EMR Serverless

To submit an Iceberg Spark job to EMR Serverless by using the AWS CLI:

1. Create the file `emr_serverless_iceberg.json` with the following content on your workstation:

```

{
  "applicationId": "<APPLICATION_ID>",
  "executionRoleArn": "<ROLE_ARN>",
  "name": "iceberg-test-job",
  "jobDriver": {
    "sparkSubmit": {
      "entryPoint": "s3://YOUR-BUCKET-NAME/code/iceberg-job.py",
      "entryPointArguments": []
    }
  },
  "configurationOverrides": {
    "applicationConfiguration": [{
      "classification": "spark-defaults",
      "properties": {
        "spark.sql.extensions":
"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions",
        "spark.sql.catalog.<catalog_name>":
"org.apache.iceberg.spark.SparkCatalog",
        "spark.sql.catalog.<catalog_name>.type": "glue",
        "spark.sql.catalog.<catalog_name>.warehouse": "s3://YOUR-BUCKET-NAME/
YOUR-FOLDER-NAME/",
        "spark.jars": "/usr/share/aws/iceberg/lib/iceberg-spark3-runtime.jar",

        "spark.hadoop.hive.metastore.client.factory.class": "com.amazonaws.glue.catalog.metastore.AWS
      }
    }],
    "monitoringConfiguration": {
      "s3MonitoringConfiguration": {
        "logUri": "s3://YOUR-BUCKET-NAME/emr-serverless/logs/"
      }
    }
  }
}

```

2. Modify the configuration file for your specific Spark job by customizing the Iceberg configuration options that are highlighted in bold.
3. Submit the job by using the AWS CLI. Run the command in the directory where the `emr_serverless_iceberg.json` file is located:

```
aws emr-serverless start-job-run --cli-input-json file://emr_serverless_iceberg.json
```

To submit an Iceberg Spark job to EMR Serverless by using the EMR Studio console:

1. Follow the instructions in the [EMR Serverless documentation](#).
2. For **Job configuration**, use the Iceberg configuration for Spark provided for the AWS CLI and customize the highlighted fields for Iceberg. For detailed instructions, see [Using Apache Iceberg with EMR Serverless](#) in the Amazon EMR documentation.

Amazon EMR on EKS

To submit an Iceberg Spark job to Amazon EMR on EKS by using the AWS CLI:

1. Create the file `emr_eks_iceberg.json` with the following content on your workstation:

```
{
  "name": "iceberg-test-job",
  "virtualClusterId": "<VIRTUAL_CLUSTER_ID>",
  "executionRoleArn": "<ROLE_ARN>",
  "releaseLabel": "emr-6.9.0-latest",
  "jobDriver": {
    "sparkSubmitJobDriver": {
      "entryPoint": "s3://YOUR-BUCKET-NAME/code/iceberg-job.py",
      "entryPointArguments": [],
      "sparkSubmitParameters": "--jars local:///usr/share/aws/iceberg/lib/iceberg-spark3-runtime.jar"
    }
  },
  "configurationOverrides": {
    "applicationConfiguration": [{
      "classification": "spark-defaults",
      "properties": {
        "spark.sql.extensions":
"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions",
        "spark.sql.catalog.<catalog_name>":
"org.apache.iceberg.spark.SparkCatalog",
        "spark.sql.catalog.<catalog_name>.type": "glue",
        "spark.sql.catalog.<catalog_name>.warehouse": "s3://YOUR-BUCKET-NAME/YOUR-FOLDER-NAME/",
        "spark.hadoop.hive.metastore.client.factory.class":
"com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory"
      }
    }],
    "monitoringConfiguration": {
```

```
    "persistentAppUI": "ENABLED",
    "s3MonitoringConfiguration": {
      "logUri": "s3://YOUR-BUCKET-NAME/emr-serverless/logs/"
    }
  }
}
```

2. Modify the configuration file for your Spark job by customizing the Iceberg configuration options that are highlighted in bold.
3. Submit the job by using the AWS CLI. Run the following command in the directory where the `emr_eks_iceberg.json` file is located:

```
aws emr-containers start-job-run --cli-input-json file://emr_eks_iceberg.json
```

For detailed instructions, see [Using Apache Iceberg with Amazon EMR on EKS](#) in the Amazon EMR on EKS documentation.

Best practices for Amazon EMR

This section provides general guidelines for tuning Spark jobs in Amazon EMR to optimize reading and writing data to Iceberg tables. For Iceberg-specific best practices, see the [Best practices](#) section later in this guide.

- **Use the latest version of Amazon EMR** – Amazon EMR provides Spark optimizations out of the box with the Amazon EMR Spark runtime. AWS improves the performance of the Spark runtime engine with each new release.
- **Determine the optimal infrastructure for your Spark workloads** – Spark workloads might require different types of hardware for different job characteristics to ensure optimal performance. Amazon EMR [supports several instance types](#) (such as compute optimized, memory optimized, general purpose, and storage optimized) to cover all types of processing requirements. When you onboard new workloads, we recommend that you benchmark with general instance types such as M5 or M6g. Monitor the operating system (OS) and YARN metrics from Ganglia and Amazon CloudWatch to determine the system bottlenecks (CPU, memory, storage, and I/O) at peak load and choose appropriate hardware.
- **Tune `spark.sql.shuffle.partitions`** – Set the `spark.sql.shuffle.partitions` property to the total number of virtual cores (vCores) in your cluster or to a multiple of that

value (typically, 1 to 2 times the total number of vCores). This setting affects the parallelism of Spark when you use hash and range partitioning as the write distribution mode. It requests a shuffle before writing to organize the data, which ensures partition alignment.

- **Enable managed scaling** – For almost all use cases, we recommend that you enable managed scaling and dynamic allocation. However, if you have a workload that has a predictable pattern, we suggest that you disable automatic scaling and dynamic allocation. When managed scaling is enabled, we recommend that you use Spot Instances to reduce costs. Use Spot Instances for task nodes instead of core or master nodes. When you use Spot Instances, use instance fleets with multiple instance types per fleet to ensure spot availability.
- **Use broadcast join when possible** – Broadcast (mapside) join is the most optimal join, as long as one of your tables is small enough to fit in the memory of your smallest node (in the order of MBs) and you are performing an equi (=) join. All join types except for full outer joins are supported. A broadcast join broadcasts the smaller table as a hash table across all worker nodes in memory. After the small table has been broadcast, you cannot make changes to it. Because the hash table is locally in the Java virtual machine (JVM), it can be merged easily with the large table based on the join condition by using a hash join. Broadcast joins provide high performance because of minimal shuffle overhead.
- **Tune the garbage collector** – If garbage collection (GC) cycles are slow, consider switching from the default parallel garbage collector to G1GC for better performance. To optimize GC performance, you can fine-tune the GC parameters. To track GC performance, you can monitor it by using the Spark UI. Ideally, the GC time should be less than or equal to 1 percent of the total task runtime.

Working with Iceberg in AWS Glue

[AWS Glue](#) is a serverless data integration service that makes it easier to discover, prepare, move, and integrate data from multiple sources for analytics, machine learning (ML), and application development. One of the core capabilities of AWS Glue is its ability to perform extract, transform, and load (ETL) operations in a simple and cost-effective manner. This helps categorize your data, clean it, enrich it, and move it reliably between various data stores and data streams.

[AWS Glue jobs](#) encapsulate scripts that define transformation logic by using an [Apache Spark](#) or Python runtime. AWS Glue jobs can be run in both batch and streaming mode.

When you create Iceberg jobs in AWS Glue, depending on the version of AWS Glue, you can use either native Iceberg integration or a custom Iceberg version to attach Iceberg dependencies to the job.

Using native Iceberg integration

AWS Glue versions 3.0, 4.0, and 5.0 natively support transactional data lake formats such as Apache Iceberg, Apache Hudi, and Linux Foundation Delta Lake in AWS Glue for Spark. This integration feature simplifies the configuration steps required to start using these frameworks in AWS Glue.

To enable Iceberg support for your AWS Glue job, set the job: Choose the **Job details** tab for your AWS Glue job, scroll to **Job parameters** under **Advanced properties**, and set the key to `--datalake-formats` and its value to `iceberg`.

If you are authoring a job by using a notebook, you can configure the parameter in the first notebook cell by using the `%%configure` magic as follows:

```
%%configure
{
  "--conf" : <job-specific Spark configuration discussed later>,
  "--datalake-formats" : "iceberg"
}
```

The `iceberg` configuration for `--datalake-formats` in AWS Glue corresponds to specific Iceberg versions based on your AWS Glue version:

AWS Glue version	Default Iceberg version
5.0	1.7.1
4.0	1.0.0
3.0	0.13.1

Using a custom Iceberg version

In some situations, you might want to retain control over the Iceberg version for the job and upgrade it at your own pace. For example, upgrading to a later version can unlock access to new features and performance enhancements. To use a specific Iceberg version with AWS Glue, you can provide your own JAR files.

Before you implement a custom Iceberg version, verify compatibility with your AWS Glue environment by checking the [AWS Glue versions](#) section of the AWS Glue documentation. For example, AWS Glue 5.0 requires compatibility with Spark 3.5.4.

As an example, to run AWS Glue jobs that use Iceberg version 1.9.1, follow these steps:

1. Acquire and upload the required JAR files to Amazon S3:
 - a. Download [iceberg-spark-runtime-3.5_2.12-1.9.1.jar](#) and [iceberg-aws-bundle-1.9.1.jar](#) from the Apache Maven repository.
 - b. Upload these files to your designated S3 bucket location (for example, `s3://your-bucket-name/jars/`).
2. Set up the job parameters for your AWS Glue job as follows:
 - a. Specify the complete S3 path to both JAR files in the `--extra-jars` parameter, separating them with a comma (for example, `s3://your-bucket-name/jars/iceberg-spark-runtime-3.5_2.12-1.9.1.jar,s3://your-bucket-name/jars/iceberg-aws-bundle-1.9.1.jar`).
 - b. Do not include `iceberg` as a value for the `--datalake-formats` parameter.
 - c. If you use AWS Glue 5.0, you must set the `--user-jars-first` parameter to `true`.

Spark configurations for Iceberg in AWS Glue

This section discusses the Spark configurations required to author an AWS Glue ETL job for an Iceberg dataset. You can set these configurations by using the `--conf` Spark key with a comma-separated list of all Spark configuration keys and values. You can use the `%%configure` magic in a notebook, or the **Job parameters** section of the AWS Glue Studio console.

```
%glue_version 5.0

%%configure
{
  "--conf" : "spark.sql.extensions=org.apache.iceberg.spark.extensions...",
  "--datalake-formats" : "iceberg"
}
```

Configure the Spark session with the following properties:

- `<catalog_name>` is the name of your Iceberg Spark session catalog name. Replace it with a name of your choice, and remember to change the references throughout all configurations that are associated with this catalog. In your code, you can refer to your Iceberg tables by using the fully qualified table name, including the Spark session catalog name, as follows:

```
<catalog_name>.<database_name>.<table_name>
```

Alternatively, you can change the default catalog to the Iceberg catalog that you defined by setting `spark.sql.defaultCatalog` to your catalog name. You can use this second approach to refer to tables without the catalog prefix, which can simplify your queries.

- `<catalog_name>.<warehouse>` points to the Amazon S3 path where you want to store your data and metadata.
- To make the catalog an AWS Glue Data Catalog, set `spark.sql.catalog.<catalog_name>.type` to `glue`. This key is required to point to an implementation class for any custom catalog implementation. For catalogs supported by Iceberg, see the [General best practices](#) section later in this guide.

For example, if you have a catalog called `glue_iceberg`, you can configure your job by using multiple `--conf` keys as follows:

```
%%configure
```

```
{
  "--datalake-formats" : "iceberg",
  "--conf" :
  "spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions
  --conf spark.sql.catalog.glue_iceberg=org.apache.iceberg.spark.SparkCatalog --
  conf spark.sql.catalog.glue_iceberg.warehouse=s3://<your-warehouse-dir>/ --conf
  spark.sql.catalog.glue_iceberg.type=glue"
}
```

Alternatively, you can use code to add the above configurations to your Spark script as follows:

```
spark = SparkSession.builder\

.config("spark.sql.extensions", "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions")\
.config("spark.sql.catalog.glue_iceberg",
"org.apache.iceberg.spark.SparkCatalog")\
.config("spark.sql.catalog.glue_iceberg.warehouse", "s3://<your-warehouse-dir>/")\
.config("spark.sql.catalog.glue_iceberg.type", "glue") \
.getOrCreate()
```

Best practices for AWS Glue jobs

This section provides general guidelines for tuning Spark jobs in AWS Glue to optimize reading and writing data to Iceberg tables. For Iceberg-specific best practices, see the [Best practices](#) section later in this guide.

- **Use the latest version of AWS Glue and upgrade whenever possible** – New versions of AWS Glue provide performance improvements, reduced startup times, and new features. They also support newer Spark versions that might be required for the latest Iceberg versions. For a list of available AWS Glue versions and the Spark versions they support, see the [AWS Glue documentation](#).
- **Optimize AWS Glue job memory** – Follow the recommendations in the AWS blog post [Optimize memory management in AWS Glue](#).
- **Use AWS Glue Auto Scaling** – When you enable Auto Scaling, AWS Glue automatically adjusts the number of AWS Glue workers dynamically based on your workload. This helps reduce the cost of your AWS Glue job during peak loads, because AWS Glue scales down the number of workers when the workload is small and workers are sitting idle. To use AWS Glue Auto Scaling,

you specify a maximum number of workers that your AWS Glue job can scale to. For more information, see [Using auto scaling for](#) AWS Glue in the AWS Glue documentation.

- **Use the desired Iceberg version** – AWS Glue native integration for Iceberg is best for getting started with Iceberg. However, for production workloads, we recommend that you add library dependencies (as discussed [earlier in this guide](#)) to get full control over the Iceberg version. This approach helps you benefit from the latest Iceberg features and performance improvements in your AWS Glue jobs.
- **Enable the Spark UI for monitoring and debugging** – You can also use the [Spark UI in AWS Glue](#) to inspect your Iceberg job by visualizing the different stages of a Spark job in a directed acyclic graph (DAG) and monitoring the jobs in detail. Spark UI provides an effective way to both troubleshoot and optimize Iceberg jobs. For example, you can identify bottleneck stages that have large shuffles or disk spill to identify tuning opportunities. For more information, see [Monitoring jobs using the Apache Spark web UI](#) in the AWS Glue documentation.

Working with Iceberg tables by using Apache Spark

This section provides an overview of using Apache Spark to interact with Iceberg tables. The examples are boilerplate code that can run on Amazon EMR or AWS Glue.

Note: The primary interface for interacting with Iceberg tables is SQL, so most of the examples will combine Spark SQL with the DataFrames API.

Creating and writing Iceberg tables

You can use Spark SQL and Spark DataFrames to create and add data to Iceberg tables.

Using Spark SQL

To write an Iceberg dataset, use standard Spark SQL statements such as `CREATE TABLE` and `INSERT INTO`.

Unpartitioned tables

Here's an example of creating an unpartitioned Iceberg table with Spark SQL:

```
spark.sql(f"""
    CREATE TABLE IF NOT EXISTS {CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_nopartitions (
        c_customer_sk          int,
        c_customer_id          string,
        c_first_name           string,
        c_last_name            string,
        c_birth_country        string,
        c_email_address        string)
    USING iceberg
    OPTIONS ('format-version'='2')
    """)
```

To insert data into an unpartitioned table, use a standard `INSERT INTO` statement:

```
spark.sql(f"""
INSERT INTO {CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_nopartitions
SELECT c_customer_sk, c_customer_id, c_first_name, c_last_name, c_birth_country,
       c_email_address
```

```
FROM another_table  
""")
```

Partitioned tables

Here's an example of creating a partitioned Iceberg table with Spark SQL:

```
spark.sql(f"""  
    CREATE TABLE IF NOT EXISTS {CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_withpartitions (  
        c_customer_sk          int,  
        c_customer_id         string,  
        c_first_name          string,  
        c_last_name           string,  
        c_birth_country       string,  
        c_email_address       string)  
    USING iceberg  
    PARTITIONED BY (c_birth_country)  
    OPTIONS ('format-version'='2')  
""")
```

To insert data into a partitioned Iceberg table with Spark SQL, use a standard `INSERT INTO` statement:

```
spark.sql(f"""  
INSERT INTO {CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_withpartitions  
SELECT c_customer_sk, c_customer_id, c_first_name, c_last_name, c_birth_country,  
       c_email_address  
FROM another_table  
""")
```

Note

Starting with Iceberg 1.5.0, hash write distribution mode is the default when you insert data into partitioned tables. For more information, see [Writing Distribution Modes](#) in the Iceberg documentation.

Using the DataFrames API

To write an Iceberg dataset, you can use the `DataFrameWriterV2` API.

To create an Iceberg table and write data to it, use the `df.writeTo(t)` function. If the table exists, use the `.append()` function. If it doesn't, use `.create()`. The following examples use `.createOrReplace()`, which is a variation of `.create()` that's equivalent to `CREATE OR REPLACE TABLE AS SELECT`.

Unpartitioned tables

To create and populate an unpartitioned Iceberg table by using the `DataFrameWriterV2` API:

```
input_data.writeTo(f"{CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_nopartitions") \  
    .tableProperty("format-version", "2") \  
    .createOrReplace()
```

To insert data into an existing unpartitioned Iceberg table by using the `DataFrameWriterV2` API:

```
input_data.writeTo(f"{CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_nopartitions") \  
    .append()
```

Partitioned tables

To create and populate a partitioned Iceberg table by using the `DataFrameWriterV2` API:

```
input_data.writeTo(f"{CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_withpartitions") \  
    .tableProperty("format-version", "2") \  
    .partitionedBy("c_birth_country") \  
    .createOrReplace()
```

To insert data into a partitioned Iceberg table by using the `DataFrameWriterV2` API:

```
input_data.writeTo(f"{CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}_withpartitions") \  
    .append()
```

Updating data in Iceberg tables

The following example shows how to update data in an Iceberg table. This example modifies all rows that have an even number in the `c_customer_sk` column.

```
spark.sql(f"""
```

```
UPDATE {CATALOG_NAME}.{db.name}.{table.name}
SET c_email_address = 'even_row'
WHERE c_customer_sk % 2 == 0
""")
```

This operation uses the default copy-on-write strategy, so it rewrites all impacted data files.

Upserting data in Iceberg tables

Upserting data refers to inserting new data records and updating existing data records in a single transaction. To upsert data into an Iceberg table, you use the SQL `MERGE INTO` statement.

The following example upserts the content of the table `{UPSERT_TABLE_NAME}` inside the table `{TABLE_NAME}`:

```
spark.sql(f"""
MERGE INTO {CATALOG_NAME}.{DB_NAME}.{TABLE_NAME} t
USING {UPSERT_TABLE_NAME} s
    ON t.c_customer_id = s.c_customer_id
    WHEN MATCHED THEN UPDATE SET t.c_email_address = s.c_email_address
    WHEN NOT MATCHED THEN INSERT *
""")
```

- If a customer record that's in `{UPSERT_TABLE_NAME}` already exists in `{TABLE_NAME}` with the same `c_customer_id`, the `{UPSERT_TABLE_NAME}` record `c_email_address` value overrides the existing value (update operation).
- If a customer record that's in `{UPSERT_TABLE_NAME}` doesn't exist in `{TABLE_NAME}`, the `{UPSERT_TABLE_NAME}` record is added to `{TABLE_NAME}` (insert operation).

Deleting data in Iceberg tables

To delete data from an Iceberg table, use the `DELETE FROM` expression and specify a filter that matches the rows to delete.

```
spark.sql(f"""
DELETE FROM {CATALOG_NAME}.{db.name}.{table.name}
WHERE c_customer_sk % 2 != 0
""")
```

If the filter matches an entire partition, Iceberg performs a metadata-only delete and leaves the data files in place. Otherwise, it rewrites only the affected data files.

The delete method takes the data files that are impacted by the `WHERE` clause and creates a copy of them without the deleted records. It then creates a new table snapshot that points to the new data files. Therefore, the deleted records are still present in the older snapshots of the table. For example, if you retrieve the previous snapshot of the table, you'll see the data that you just deleted. For information about removing unneeded old snapshots with the related data files for cleanup purposes, see the section [Maintaining files by using compaction](#) later in this guide.

Reading data

You can read the latest status of your Iceberg tables in Spark with both Spark SQL and DataFrames.

Example using Spark SQL:

```
spark.sql(f"""
SELECT * FROM {CATALOG_NAME}.{db.name}.{table.name} LIMIT 5
""")
```

Example using the DataFrames API:

```
df = spark.table(f"{CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}").limit(5)
```

Using time travel

Each write operation (insert, update, upsert, delete) in an Iceberg table creates a new snapshot. You can then use these snapshots for time travel—to go back in time and check the status of a table in the past.

For information about how to retrieve the history of snapshots for tables by using `snapshot-id` and timing values, see the [Accessing metadata](#) section later in this guide.

The following time travel query displays the status of a table based on a specific `snapshot-id`.

Using Spark SQL:

```
spark.sql(f"""
```

```
SELECT * FROM {CATALOG_NAME}.{DB_NAME}.{TABLE_NAME} VERSION AS OF {snapshot_id}
""")
```

Using the DataFrames API:

```
df_1st_snapshot_id = spark.read.option("snapshot-id", snapshot_id) \
    .format("iceberg") \
    .load(f"{CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}") \
    .limit(5)
```

The following time travel query displays the status of a table based on the last snapshot that was created before a specific timestamp, in milliseconds (`as-of-timestamp`).

Using Spark SQL:

```
spark.sql(f"""
SELECT * FROM dev.{db.name}.{table.name} TIMESTAMP AS OF '{snapshot_ts}'
""")
```

Using the DataFrames API:

```
df_1st_snapshot_ts = spark.read.option("as-of-timestamp", snapshot_ts) \
    .format("iceberg") \
    .load(f"dev.{DB_NAME}.{TABLE_NAME}") \
    .limit(5)
```

Using incremental queries

You can also use Iceberg snapshots to read appended data incrementally.

Note: Currently, this operation supports reading data from append snapshots. It doesn't support fetching data from operations such as `replace`, `overwrite`, or `delete`. Additionally, incremental read operations aren't supported in the Spark SQL syntax.

The following example retrieves all the records appended to an Iceberg table between the snapshot `start-snapshot-id` (exclusive) and `end-snapshot-id` (inclusive).

```
df_incremental = (spark.read.format("iceberg")
    .option("start-snapshot-id", snapshot_id_start)
```

```

.option("end-snapshot-id", snapshot_id_end)
.load(f"glue_catalog.{DB_NAME}.{TABLE_NAME}")
)

```

Accessing metadata

Iceberg provides access to its metadata through SQL. You can access the metadata for any given table (<table_name>) by querying the namespace <table_name>.<metadata_table>. For a complete list of metadata tables, see [Inspecting tables](#) in the Iceberg documentation.

The following example shows how to access the Iceberg history metadata table, which shows the history of commits (changes) for an Iceberg table.

Using Spark SQL (with the %%sql magic) from an Amazon EMR Studio notebook:

```

Spark.sql(f"""
SELECT * FROM {CATALOG_NAME}.{DB_NAME}.{TABLE_NAME}.history LIMIT 5
""")

```

Using the DataFrames API:

```

spark.read.format("iceberg").load("{CATALOG_NAME}.{DB_NAME}."
{TABLE_NAME}.history").show(5, False)

```

Sample output:

Type:	Table	Pie	Scatter	Line	Area	Bar
	made_current_at	snapshot_id	parent_id	is_current_ancestor		
	2023-01-09 02:50:17.547000+00:00	7501027970051178613	6598755163776233735	True		
	2023-01-12 05:39:29.567000+00:00	7069175828427777019	7501027970051178613	True		
	2023-01-12 05:39:58.807000+00:00	5173022175861138222	7069175828427777019	True		
	2023-01-12 05:40:18.499000+00:00	3703414997660223390	5173022175861138222	True		
	2023-01-12 05:40:41.827000+00:00	3807904412292252460	3703414997660223390	True		

Working with Iceberg tables by using Trino

This section describes how to set up and operate Iceberg tables by using [Trino](#) on [Amazon EMR](#). The examples are boilerplate code that you can run on an Amazon EMR on EC2 cluster. The code examples and configurations in this section assume that you're using Amazon EMR release **emr-7.9.0**.

Amazon EMR on EC2 setup

1. Create an `iceberg.properties` file with the following content. The `iceberg.file-format=parquet` setting determines the default storage format for new tables if the format isn't explicitly specified in the `CREATE TABLE` statement.

```
connector.name=iceberg
iceberg.catalog.type=glue
iceberg.file-format=parquet
fs.native-s3.enabled=true
```

2. Upload the `iceberg.properties` file to your S3 bucket.
3. Create a bootstrap action that copies the `iceberg.properties` file from your S3 bucket and stores it as a Trino configuration file on the Amazon EMR cluster that you will be creating. Make sure to replace `<S3-bucket-name>` with your S3 bucket name.

```
#!/bin/bash
set -ex
sudo aws s3 cp s3://<S3-bucket-name>/iceberg.properties /etc/trino/conf/catalog/
iceberg.properties
```

4. Create an Amazon EMR cluster with Trino installed and specify the execution of the previous script as a bootstrap action. Here's a sample AWS Command Line Interface (AWS CLI) command for creating the cluster:

```
aws emr create-cluster --release-label emr-7.9.0 \
--applications Name=Trino \
--region <region> \
--name Trino_Iceberg_Cluster \
--bootstrap-actions '[{"Path":"s3://<S3-bucket-name>/bootstrap.sh", "Name":"Add
iceberg.properties"}]' \
```

```
--instance-groups
' [{"InstanceGroupType": "MASTER", "InstanceCount": 1, "InstanceType": "m5.xlarge"},
{"InstanceGroupType": "CORE", "InstanceCount": 3, "InstanceType": "m5.xlarge"} ]' \
--service-role "<IAM-service-role>" \
--ec2-attributes '{"KeyName": "<key-name>", "InstanceProfile": "<EMR-EC2-instance-
profile>"}'
```

where you replace:

- <S3-bucket-name> with your S3 bucket name
 - <region> with your specific AWS Region
 - <key-name> with your key pair. If the key pair doesn't exist, it will be created.
 - <IAM-service-role> with your Amazon EMR service role that follows the [principle of least privilege](#).
 - <EMR-EC2-instance-profile> with your [instance profile](#).
5. When the Amazon EMR cluster has been initialized, you can initialize a Trino session by running the following command:

```
trino-cli
```

6. In the Trino CLI, you can view the catalogs by running:

```
SHOW CATALOGS;
```

Creating Iceberg tables

To create an Iceberg table, you can use the CREATE TABLE statement. Here's an example of creating a partitioned table that uses Iceberg hidden partitioning:

```
CREATE TABLE iceberg.iceberg_db.iceberg_table (
    userid int,
    firstname varchar,
    city varchar)
WITH (
    format = 'PARQUET',
    partitioning = ARRAY['city', 'bucket(userid, 16)'],
    location = 's3://<S3-bucket>/<prefix>');
```

Note

If you don't specify the format, the `iceberg.file-format` value that you configured in the previous section will be used.

To insert data, use the `INSERT INTO` command. Here's an example:

```
INSERT INTO iceberg.iceberg_db.iceberg_table (userid, firstname, city)
VALUES
  (1001, 'John', 'New York'),
  (1002, 'Mary', 'Los Angeles'),
  (1003, 'Mateo', 'Chicago'),
  (1004, 'Shirley', 'Houston'),
  (1005, 'Diego', 'Miami'),
  (1006, 'Nikki', 'Seattle'),
  (1007, 'Pat', 'Boston'),
  (1008, 'Terry', 'San Francisco'),
  (1009, 'Richard', 'Denver'),
  (1010, 'Pat', 'Phoenix');
```

Reading from Iceberg tables

You can read the latest status of your Iceberg table by using a `SELECT` statement, as follows:

```
SELECT * FROM iceberg.iceberg_db.iceberg_table;
```

Upserting data into Iceberg tables

You can perform an upsert operation (simultaneously insert new records and update existing ones) by using the `MERGE INTO` statement. Here's an example:

```
MERGE INTO iceberg.iceberg_db.iceberg_table target
USING (
  VALUES
    (1001, 'John Updated', 'Boston'),      -- Update existing user
    (1002, 'Mary Updated', 'Seattle'),    -- Update existing user
    (1011, 'Martha', 'Portland'),         -- Insert new user
    (1012, 'Paulo', 'Austin')            -- Insert new user
```

```
) AS source (userid, firstname, city)
ON target.userid = source.userid
WHEN MATCHED THEN
  UPDATE SET
    firstname = source.firstname,
    city = source.city
WHEN NOT MATCHED THEN
  INSERT (userid, firstname, city)
  VALUES (source.userid, source.firstname, source.city);
```

Deleting records from Iceberg tables

To delete data from an Iceberg table, use the `DELETE FROM` expression and specify a filter that matches the rows to delete. Here's an example:

```
DELETE FROM iceberg.iceberg_db.iceberg_table WHERE userid IN (1003, 1004);
```

Querying Iceberg table metadata

Iceberg provides access to its metadata through SQL. You can access the metadata for any given table (`<table_name>`) by querying the namespace "`<table_name>.$<metadata_table>`". For a complete list of metadata tables, see [Inspecting tables](#) in the Iceberg documentation.

Here's an example list of queries to inspect Iceberg metadata:

```
SELECT FROM iceberg.iceberg_db."iceberg_table$snapshots";
SELECT FROM iceberg.iceberg_db."iceberg_table$history";
SELECT FROM iceberg.iceberg_db."iceberg_table$partitions";
SELECT FROM iceberg.iceberg_db."iceberg_table$files";
SELECT FROM iceberg.iceberg_db."iceberg_table$manifests";
SELECT FROM iceberg.iceberg_db."iceberg_table$refs";
SELECT * FROM iceberg.iceberg_db."iceberg_table$metadata_log_entries";
```

For example, this query:

```
SELECT * FROM iceberg.iceberg_db."iceberg_table$snapshots";
```

provides the output:

```
trino> SELECT * FROM iceberg.iceberg_db."iceberg_table$snapshots";
  committed_at | snapshot_id | parent_id | operation | manifest_list
-----|-----|-----|-----|-----
2025-05-28 16:05:41.801 UTC | 7785073462465010154 | NULL | append | s3://
2025-05-28 16:05:57.806 UTC | 5984821362426775846 | 7785073462465010154 | append | s3://
2025-05-28 16:09:40.268 UTC | 241938428756831817 | 5984821362426775846 | overwrite | s3://
2025-05-28 16:18:53.126 UTC | 1784832837567742464 | 241938428756831817 | delete | s3://
(4 rows)

Query 20250528_162032_00012_uhduz, FINISHED, 1 node
Splits: 1 total, 1 done (100.00%)
0.30 [4 rows, 3.11KiB] [13 rows/s, 10.3KiB/s]
```

Using time travel

Each write operation (insert, update, upsert, or delete) in an Iceberg table creates a new snapshot. You can then use these snapshots for time travel—to go back in time and check the status of a table in the past.

The following time travel query displays the status of a table based on a specific `snapshot_id`:

```
SELECT *
FROM iceberg.iceberg_db.iceberg_table FOR VERSION AS OF 241938428756831817;
```

The following time travel query displays the status of a table based on a specific timestamp:

```
SELECT *
FROM iceberg.iceberg_db.iceberg_table FOR TIMESTAMP AS OF TIMESTAMP '2025-05-28
16:09:40.268 UTC'
```

Considerations when using Iceberg with Trino

Trino write operations on Iceberg tables follow the [merge-on-read](#) design, so they create positional delete files instead of rewriting entire data files that are impacted by updates or deletes. If you want to use the copy-on-write approach, consider using Spark for write operations.

Working with Iceberg tables by using Amazon Data Firehose

Amazon Data Firehose is a serverless, no-code service for delivering data streams from over 20 sources such as AWS WAF logs, Amazon CloudWatch Logs, AWS IoT, Amazon Kinesis Data Streams, and Amazon Managed Streaming for Apache Kafka (Amazon MSK) into destinations such as Amazon S3, Amazon Redshift, Snowflake, and Splunk.

You can use Firehose to directly deliver streaming data to Apache Iceberg tables in Amazon S3. Using Firehose, you can route records from a single stream into different Apache Iceberg tables, and automatically apply insert, update, and delete operations to records in the tables. Firehose guarantees exactly-once delivery to Iceberg tables. This feature requires using the AWS Glue Data Catalog.

Firehose can also directly deliver streaming data to Amazon S3 tables. These tables provide storage that is optimized for large-scale analytics workloads, and include features that continuously improve query performance and reduce storage costs for tabular data.

For information about how to set up a Firehose stream to deliver data to Apache Iceberg tables, see [Set up the Firehose stream](#) in the Firehose documentation or the blog post [Stream real-time data into Apache Iceberg tables in Amazon S3 using Amazon Data Firehose](#).

Working with Iceberg tables by using Athena SQL

Amazon Athena provides built-in support for Apache Iceberg, and doesn't require additional steps or configuration. This section provides a detailed overview of supported features and high-level guidance for using Athena to interact with Iceberg tables.

Version and feature compatibility

Iceberg table specification support

The Apache Iceberg table specification specifies how Iceberg tables should behave. Athena supports table format version 2, so any Iceberg table that you create with the console, CLI, or SDK inherently uses that version.

If you use an Iceberg table that was created with another engine, such as Apache Spark on Amazon EMR or AWS Glue, make sure to set the table format version by using [table properties](#). As a reference, see the section [Creating and writing Iceberg tables](#) earlier in this guide.

Iceberg feature support

You can use Athena to read from and write to Iceberg tables. When you change data by using the UPDATE, MERGE INTO, and DELETE FROM statements, Athena supports merge-on-read mode only. This property cannot be changed. In order to update or delete data with copy-on-write, you have to use other engines such as Apache Spark on Amazon EMR or AWS Glue. The following table summarizes Iceberg feature support in Athena.

		DDL support		DML support		AWS Lake Formation for security (optional)
	Table format	Create table	Schema evolution	Reading data	Writing data	Row/column access control

		DDL support		DML support		AWS Lake Formation for security (optional)
Amazon Athena	Version 2	✓	✓	✓	X Copy-on-write	✓
					✓ Merge-on-read	✓

Note

- Athena doesn't support incremental queries.
- In Athena, update, delete, and merge operations always default to merge on read (MoR), regardless of any copy on write (CoW) settings in the table properties, because CoW isn't supported.

Working with Iceberg tables

For a quick start to using Iceberg in Athena, see the section [Getting started with Iceberg tables in Athena SQL](#) earlier in this guide.

The following table lists limitations and recommendations.

Scenario	Limitation	Recommendation
Table DDL generation	Iceberg tables created with other engines can have properties that are not exposed in Athena. For these tables, it's not possible to generate the DDL.	Use the equivalent statement in the engine that created the table (for example, the SHOW CREATE TABLE statement for Spark).

Scenario	Limitation	Recommendation
Random Amazon S3 prefixes in objects that are written to an Iceberg table	By default, Iceberg tables that are created with Athena have the <code>write.object-storage.enabled</code> property enabled.	To disable this behavior and gain full control over Iceberg table properties, create an Iceberg table with another engine such as Spark on Amazon EMR or AWS Glue.
Incremental queries	Not currently supported in Athena.	To use incremental queries to enable incremental data ingestion pipelines, use Spark on Amazon EMR or AWS Glue.

Working with Iceberg tables by using Pylceberg

This section explains how you can interact with Iceberg tables by using [Pylceberg](#). The examples provided are boilerplate code that you can run on [Amazon Linux 2023 EC2](#) instances, [AWS Lambda](#) functions, or any [Python](#) environment with properly configured [AWS credentials](#).

Prerequisites

Note

These examples use [Pylceberg 1.9.1](#).

To work with Pylceberg, you need Pylceberg and AWS SDK for Python (Boto3) installed. Here's an example of how you can set up a Python virtual environment to work with Pylceberg and AWS Glue Data Catalog:

1. Download [Pylceberg](#) by using the [pip python package installer](#). You also need [Boto3](#) to interact with AWS services. You can configure a local Python virtual environment to test by using these commands:

```
python3 -m venv my_env
cd my_env/bin/
source activate
pip install "pyiceberg[pyarrow,pandas,glue]"
pip install boto3
```

2. Run python to open the Python shell and test the commands.

Connecting to the Data Catalog

To start working with Iceberg tables in AWS Glue, you first need to connect to the AWS Glue Data Catalog.

The `load_catalog` function initializes a connection to the Data Catalog by creating a [catalog](#) object that serves as your primary interface for all Iceberg operations:

```
from pyiceberg.catalog import load_catalog
```

```
region = "us-east-1"

glue_catalog = load_catalog(
    'default',
    **{
        'client.region': region
    },
    type='glue'
)
```

Listing and creating databases

To list existing databases, use the `list_namespaces` function:

```
databases = glue_catalog.list_namespaces()
print(databases)
```

To create a new database, use the `create_namespace` function:

```
database_name="mydb"
s3_db_path=f"s3://amzn-s3-demo-bucket/{database_name}"

glue_catalog.create_namespace(database_name, properties={"location": s3_db_path})
```

Creating and writing Iceberg tables

Unpartitioned tables

Here's an example of creating an unpartitioned Iceberg table by using the `create_table` function:

```
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, DoubleType

database_name="mydb"
table_name="pyiceberg_table"
s3_table_path=f"s3://amzn-s3-demo-bucket/{database_name}/{table_name}"

schema = Schema(
```

```
NestedField(1, "city", StringType(), required=False),
NestedField(2, "lat", DoubleType(), required=False),
NestedField(3, "long", DoubleType(), required=False),
)

glue_catalog.create_table(f"{database_name}.{table_name}", schema=schema,
location=s3_table_path)
```

You can use the `list_tables` function to check the list of tables inside a database:

```
tables = glue_catalog.list_tables(namespace=database_name)
print(tables)
```

You can use the `append` function and `PyArrow` to insert data inside an Iceberg table:

```
import pyarrow as pa
df = pa.Table.from_pylist(
    [
        {"city": "Amsterdam", "lat": 52.371807, "long": 4.896029},
        {"city": "San Francisco", "lat": 37.773972, "long": -122.431297},
        {"city": "Drachten", "lat": 53.11254, "long": 6.0989},
        {"city": "Paris", "lat": 48.864716, "long": 2.349014},
    ],
)

table = glue_catalog.load_table(f"{database_name}.{table_name}")
table.append(df)
```

Partitioned tables

Here's an example of creating a [partitioned](#) Iceberg table with [hidden partitioning](#) by using the `create_table` function and `PartitionSpec`:

```
from pyiceberg.schema import Schema
from pyiceberg.types import (
    NestedField,
    StringType,
    FloatType,
    DoubleType,
    TimestampType,
)
```

```

# Define the schema
schema = Schema(
    NestedField(field_id=1, name="datetime", field_type=TimestampType(),
required=True),
    NestedField(field_id=2, name="drone_id", field_type=StringType(), required=True),
    NestedField(field_id=3, name="lat", field_type=DoubleType(), required=False),
    NestedField(field_id=4, name="lon", field_type=DoubleType(), required=False),
    NestedField(field_id=5, name="height", field_type=FloatType(), required=False),
)

from pyiceberg.partitioning import PartitionSpec, PartitionField
from pyiceberg.transforms import DayTransform

partition_spec = PartitionSpec(
    PartitionField(
        source_id=1, # Refers to "datetime"
        field_id=1000,
        transform=DayTransform(),
        name="datetime_day"
    )
)

database_name="mydb"
partitioned_table_name="pyiceberg_table_partitioned"
s3_table_path=f"s3://amzn-s3-demo-bucket/{database_name}/{partitioned_table_name}"

glue_catalog.create_table(
    identifier=f"{database_name}.{partitioned_table_name}",
    schema=schema,
    location=s3_table_path,
    partition_spec=partition_spec
)

```

You can insert data into a partitioned table the same way as for an unpartitioned table. The partitioning is handled automatically.

```

from datetime import datetime
arrow_schema = pa.schema([
    pa.field("datetime", pa.timestamp("us"), nullable=False),
    pa.field("drone_id", pa.string(), nullable=False),
    pa.field("lat", pa.float64()),
    pa.field("lon", pa.float64()),

```

```
    pa.field("height", pa.float32()),
])

data = [
    {
        "datetime": datetime(2024, 6, 1, 12, 0, 0),
        "drone_id": "drone_001",
        "lat": 52.371807,
        "lon": 4.896029,
        "height": 120.5,
    },
    {
        "datetime": datetime(2024, 6, 1, 12, 5, 0),
        "drone_id": "drone_002",
        "lat": 37.773972,
        "lon": -122.431297,
        "height": 150.0,
    },
    {
        "datetime": datetime(2024, 6, 2, 9, 0, 0),
        "drone_id": "drone_001",
        "lat": 53.11254,
        "lon": 6.0989,
        "height": 110.2,
    },
    {
        "datetime": datetime(2024, 6, 2, 9, 30, 0),
        "drone_id": "drone_003",
        "lat": 48.864716,
        "lon": 2.349014,
        "height": 145.7,
    },
]

df = pa.Table.from_pylist(data, schema=arrow_schema)

table = glue_catalog.load_table(f"{database_name}.{partitioned_table_name}")
table.append(df)
```

Reading data

You can use the Pylceberg scan function to read data from your Iceberg tables. You can filter rows, select specific columns, and limit the number of returned records.

```
table= glue_catalog.load_table(f"{database_name}.{table_name}")
scan_df = table.scan(
    row_filter=(
        f"city = 'Amsterdam'"
    ),
    selected_fields=("city", "lat"),
    limit=100,
).to_pandas()

print(scan_df)
```

Deleting data

The Pylceberg delete function lets you remove records from your table by using a `delete_filter`:

```
table = glue_catalog.load_table(f"{database_name}.{table_name}")
table.delete(delete_filter="city == 'Paris'")
```

Accessing metadata

Pylceberg provides several functions to access table metadata. Here's how you can view information about table snapshots:

```
#List of snapshots
table.snapshots()

#Current snapshot
table.current_snapshot()

#Take a previous snapshot
second_last_snapshot_id=table.snapshots()[-2].snapshot_id
print(f"Second last SnapshotID: {second_last_snapshot_id}")
```

For a detailed list of available metadata, see the [metadata](#) code reference section of the Pylceberg documentation.

Using time travel

You can use table snapshots for time travel to access previous states of your table. Here's how to view the table state before the last operation:

```
second_last_snapshot_id=table.snapshots()[-2].snapshot_id

time_travel_df = table.scan(
    limit=100,
    snapshot_id=second_last_snapshot_id
).to_pandas()

print(time_travel_df)
```

For a complete list of available functions, see the Pylceberg [Python API](#) documentation.

Working with Iceberg table format specification version 3

The latest version of the Apache Iceberg table format specification is version 3. This version introduces advanced capabilities for building petabyte-scale data lakes with improved performance and reduced operational overhead. It addresses common performance bottlenecks encountered with version 2, particularly around batch updates and compliance delete operations.

AWS provides support for deletion vectors and row lineage as defined in the Iceberg version 3 specification. These features are available with Apache Spark on the following AWS services.

AWS service	Version 3 support
Amazon EMR for Apache Spark	Amazon EMR release 7.12 and later
AWS Glue	Yes
AWS Glue: Iceberg REST API , table maintenance	Yes
Amazon SageMaker Unified Studio notebooks	Yes
Amazon S3 Tables: Iceberg REST API , table maintenance	Yes
Amazon Athena (Trino)	No

Key features in version 3

Deletion vectors replace the positional delete files that were used in version 2 with an efficient binary format stored as Puffin files. This eliminates write amplification from random batch updates and General Data Protection Regulation (GDPR) compliance deletes, and significantly reduces the overhead of maintaining fresh data. Organizations that process high-frequency updates will see immediate improvements in write performance and reduced storage costs from fewer small files.

Row lineage enables precise change tracking at the row level. Your downstream systems can process changes incrementally, speeding up data pipelines and reducing compute costs for change

data capture (CDC) workflows. This built-in capability eliminates the need for custom change tracking implementations.

Version compatibility

Version 3 maintains backward compatibility with version 2 tables. AWS services support both version 2 and version 3 tables simultaneously, so you can:

- Run queries across both version 2 and version 3 tables.
- Upgrade existing version 2 tables to version 3 without data rewrites.
- Run time travel queries that span version 2 and version 3 snapshots.
- Use schema evolution and hidden partitioning across table versions.

Getting started with version 3

Prerequisites

Before working with version 3 tables, make sure that you have:

- An AWS account with appropriate AWS Identity and Access Management (IAM) permissions.
- Access to one or more AWS analytics services (Amazon EMR, AWS Glue, Amazon SageMaker Unified Studio notebooks, or Amazon S3 Tables).
- An S3 bucket for storing table data and metadata.
- A table bucket to get started with Amazon S3 Tables or a general-purpose S3 bucket if you are building your own Iceberg infrastructure.
- A configured AWS Glue catalog.

Creating version 3 tables

Creating new tables

To create a new Iceberg version 3 table, set the `format-version` table property to 3.

Using Spark SQL:

```
CREATE TABLE IF NOT EXISTS myns.orders_v3 (
```

```
order_id bigint,  
customer_id string,  
order_date date,  
total_amount decimal(10,2),  
status string,  
created_at timestamp  
)  
USING iceberg  
TBLPROPERTIES (  
  'format-version' = '3'  
)
```

Upgrading version 2 tables to version 3

You can upgrade existing version 2 tables to version 3 atomically without rewriting data.

Using Spark SQL:

```
ALTER TABLE myns.existing_table  
SET TBLPROPERTIES ('format-version' = '3')
```

Important

Version 3 is a one-way upgrade. After a table is upgraded from version 2 to version 3, it cannot be downgraded back to version 2 through standard operations.

What happens during upgrade:

- A new metadata snapshot is created atomically.
- Existing Parquet data files are reused.
- Row lineage fields are added to the table metadata.

After the upgrade:

- The next compaction will remove old version 2 delete files.
- New modifications will use the version 3 deletion vector files.

The upgrade doesn't perform a historical backfill of row lineage change tracking records.

Enabling deletion vectors

To take advantage of deletion vectors for updates, deletes, and merges, configure your write mode.

Using Spark SQL:

```
ALTER TABLE myns.orders_v3
SET TBLPROPERTIES ('format-version' = '3',
                  'write.delete.mode' = 'merge-on-read',
                  'write.update.mode' = 'merge-on-read',
                  'write.merge.mode' = 'merge-on-read'
                  )
```

These settings ensure that update, delete, and merge operations create deletion vector files instead of rewriting entire data files.

Using row lineage for change tracking

Version 3 automatically adds row lineage metadata fields to track changes.

Using Spark SQL:

```
# Query with parameter value provided
last_processed_sequence = 47

SELECT
  id,
  data,
  _row_id,
  _last_updated_sequence_number
FROM myns.orders_v3
WHERE _last_updated_sequence_number > :last_processed_sequence
```

The `_row_id` field uniquely identifies each row, and `_last_updated_sequence_number` tracks when the row was last modified. Use these fields to:

- Identify changed rows for incremental processing.
- Track data lineage for compliance.
- Optimize CDC pipelines.

- Reduce compute costs by processing only changes.

Best practices for version 3

When to use version 3

Consider upgrading to, or starting with, version 3 when:

- You perform frequent batch updates or deletes.
- You need to meet GDPR or compliance delete requirements.
- Your workloads involve high-frequency upserts.
- You require efficient CDC workflows.
- You want to reduce storage costs from small files.
- You need better change tracking capabilities.

Optimizing write performance

- Enable deletion vectors for update-heavy workloads:

```
SET TBLPROPERTIES (  
  'write.delete.mode' = 'merge-on-read',  
  'write.update.mode' = 'merge-on-read',  
  'write.merge.mode' = 'merge-on-read'  
)
```

- Configure appropriate file sizes:

```
SET TBLPROPERTIES (  
  'write.target-file-size-bytes' = '536870912' -- 512 MB  
)
```

Optimizing read performance

- Use row lineage for incremental processing.
- Use time travel to access historical data without copying.
- Enable statistics collection for better query planning.

Migration strategy

When you migrate from version 2 to version 3, follow these best practices:

- Test in a non-production environment first to validate the upgrade process and performance.
- Upgrade during low-activity periods to minimize impact on concurrent operations.
- Monitor initial performance, and track metrics after the upgrade.
- Run compaction to consolidate delete files after the upgrade.
- Update your team documentation to reflect version 3 features.

Compatibility considerations

- Engine versions – Make sure that all engines accessing the table support version 3.
- Third-party tools – Verify your tool's version 3 compatibility before you upgrade.
- Backup strategy – Test snapshot-based recovery procedures.
- Monitoring – Update monitoring dashboards for version 3-specific metrics.

Troubleshooting

Common issues

Error: "format-version 3 is not supported"

- Verify that your engine version supports version 3. For specifics, see the [table](#) at the beginning of this section.
- Check catalog compatibility.
- Make sure that you're using the latest versions of AWS services.

Performance degradation after upgrade

- Verify that there are no compaction failures. For more information, see [Logging and monitoring for S3 Tables](#) in the Amazon S3 documentation.
- Confirm that deletion vectors are enabled. The following properties should be set:

```
SET TBLPROPERTIES (  
  'write.delete.mode' = 'merge-on-read',  
  'write.update.mode' = 'merge-on-read',  
  'write.merge.mode' = 'merge-on-read'  
)
```

You can verify table properties with the following code:

```
DESCRIBE FORMATTED myns.orders_v3
```

- Review your partition strategy. Over-partitioning can lead to small files. Run the following query to get the average file size for your table:

```
SELECT avg(file_size_in_bytes) as avg_file_size_bytes  
FROM myns.orders_v3.files
```

Incompatibility with third-party tools

- Verify that the tool supports the version 3 specification.
- Consider maintaining version 2 tables for unsupported tools.
- Contact the tool vendor for their version 3 support timeline.

Getting help

- For AWS service-specific issues, contact [AWS Support](#).
- To get help from the Iceberg community, use the [Iceberg Slack channel](#).
- For information about using AWS services to manage your analytics workloads, see [Analytics on AWS](#).

Pricing

- [Amazon EMR compute and storage pricing](#)
- [Amazon SageMaker pricing](#)
- [AWS Glue job run and Data Catalog pricing](#)

- [S3 Tables storage and requests pricing](#)

Availability

Iceberg table format specification version 3 support is available in all AWS Regions where Amazon EMR, AWS Glue, AWS Glue Data Catalog, and S3 Tables operate. For Region availability, see [AWS services by Region](#).

Additional resources

- [Apache Iceberg documentation](#)
- [Apache Iceberg table spec](#)
- [Guidance for migrating tabular data from Amazon S3 to S3 Tables](#)
- [Tutorial: Getting started with S3 Tables](#)

Migrating existing tables to Iceberg

This section focuses on migrating your existing Hive-style tables to Iceberg format. It applies to tables that use traditional Hive-compatible formats such as [Apache Parquet](#) or [Apache ORC](#). This information doesn't apply to tables that already use modern table formats such as Linux Foundation Delta Lake or Apache Hudi.

To migrate your current Hive-style tables to Iceberg format, you can use either in-place or full data migration:

- [In-place migration](#) is the process of generating Iceberg's metadata files on top of existing data files.
- [Full data migration](#) creates the Iceberg metadata layer and also rewrites existing data files from the original table to the new Iceberg table.

The following sections provide a detailed overview of each migration method, including step-by-step instructions and considerations for implementation. For more information about these migration strategies, see the [Table Migration](#) section of the Iceberg documentation.

After you review the details of the in-place and full data migration methods, see the following two key sections to aid your decision-making process:

- [Choosing a migration strategy](#) provides guidance through a series of questions and scenarios, to help you determine the most suitable migration approach based on your specific requirements and use cases.
- [Migration options summary](#) provides a comprehensive table that compares key characteristics and considerations across different migration options. This table serves as a quick reference guide and offers a feature comparison to help you understand the technical trade-offs between methods.

In-place migration

In-place migration eliminates the need to rewrite all your data files. Instead, Iceberg metadata files are generated and linked to your existing data files. This method is typically faster and more cost-effective, especially for large datasets or tables that have compatible file formats such as Parquet, Avro, and ORC.

Note

In-place migration cannot be used when migrating to [Amazon S3 Tables](#).

Iceberg offers two main options for implementing in-place migration:

- Using the [snapshot](#) procedure to create a new Iceberg table while keeping the source table unchanged. For more information, see [Snapshot Table](#) in the Iceberg documentation.
- Using the [migrate](#) procedure to create a new Iceberg table as a substitution for the source table. For more information, see [Migrate Table](#) in the Iceberg documentation. Although this procedure works with Hive Metastore (HMS), it isn't currently compatible with the AWS Glue Data Catalog. The [Replicating the table migration procedure in AWS Glue Data Catalog](#) section later in this guide provides a workaround for achieving a similar outcome with the Data Catalog.

After you perform in-place migration by using either `snapshot` or `migrate`, some data files might remain unmigrated. This typically happens when writers continue writing to the source table during or after migration. To incorporate these remaining files into your Iceberg table, you can use the [add_files](#) procedure. For more information, see [Add Files](#) in the Iceberg documentation.

Let's say you have a Parquet-based `products` table that was created and populated in Athena as follows:

```
CREATE EXTERNAL TABLE mydb.products (  
    product_id INT,  
    product_name STRING  
)  
PARTITIONED BY (category STRING)  
STORED AS PARQUET  
LOCATION 's3://amzn-s3-demo-bucket/products/';  
  
INSERT INTO mydb.products  
VALUES  
    (1001, 'Smartphone', 'electronics'),  
    (1002, 'Laptop', 'electronics'),  
    (2001, 'T-Shirt', 'clothing'),  
    (2002, 'Jeans', 'clothing');
```

The following sections explain how you can use the snapshot and migrate procedures with this table.

Option 1: snapshot procedure

The snapshot procedure creates a new Iceberg table that has a different name but replicates the schema and partitioning of the source table. This operation leaves the source table completely unchanged both during and after the action. It effectively creates a lightweight copy of the table, which is particularly useful for testing scenarios or data exploration without risking modifications to the original data source. This approach enables a transition period where both the original table and the Iceberg table remain available (see the notes at the end of this section). When testing is complete, you can move your new Iceberg table to production by transitioning all writers and readers to the new table.

You can run the snapshot procedure by using Spark in any Amazon EMR deployment model (for example, Amazon EMR on EC2, Amazon EMR on EKS, EMR Serverless) and AWS Glue.

To test in-place migration with the snapshot Spark procedure, follow these steps:

1. Launch a Spark application and configure the Spark session with the following settings:

- "spark.sql.extensions":"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
- "spark.sql.catalog.spark_catalog":"org.apache.iceberg.spark.SparkSessionCatalog"
- "spark.sql.catalog.spark_catalog.type":"glue"
- "spark.hadoop.hive.metastore.client.factory.class":"com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClient"

2. Run the snapshot procedure to create a new Iceberg table that points to the original table data files:

```
spark.sql(f"""
CALL system.snapshot(
  source_table => 'mydb.products',
  table => 'mydb.products_iceberg',
  location => 's3://amzn-s3-demo-bucket/products_iceberg/'
)
""")
).show(truncate=False)
```

The output dataframe contains the `imported_files_count` (the numbers of files that were added).

3. Validate the new table by querying it:

```
spark.sql(f"""
SELECT * FROM mydb.products_iceberg LIMIT 10
""")
).show(truncate=False)
```

Notes

- After you run the procedure, any data file modifications on the source table will throw the generated table out of sync. New files that you add won't be visible in the Iceberg table, and files that you removed will affect query capabilities in the Iceberg table. To avoid the synchronization issues:
 - If the new Iceberg table is intended for production use, stop all processes that write to the original table and redirect them to the new table.
 - If you need a transition period or if the new Iceberg table is for testing purposes, see [Keeping Iceberg tables in sync after in-place migration](#) later in this section for guidance on maintaining table synchronization.
- When you use the snapshot procedure, the `gc.enabled` property is set to `false` in the table properties of the created Iceberg table. This setting prohibits actions such as `expire_snapshots`, `remove_orphan_files`, or `DROP TABLE` with the `PURGE` option, which would physically delete data files. Iceberg delete or merge operations, which do not directly impact source files, are still allowed.
- To make your new Iceberg table fully functional, with no limits on actions that physically delete data files, you can change the `gc.enabled` table property to `true`. However, this setting will allow actions that impact source data files, which could corrupt access to the original table. Therefore, change the `gc.enabled` property only if you no longer need to maintain the original table's functionality. For example:

```
spark.sql(f"""
ALTER TABLE mydb.products_iceberg
SET TBLPROPERTIES ('gc.enabled' = 'true');
""")
```

Option 2: migrate procedure

The `migrate` procedure creates a new Iceberg table that has the same name, schema, and partitioning as the source table. When this procedure runs, it locks the source table and renames it to `<table_name>_BACKUP_` (or a custom name specified by the `backup_table_name` procedure parameter).

Note

If you set the `drop_backup` procedure parameter to `true`, the original table will not be retained as a backup.

Consequently, the `migrate` table procedure requires all modifications that affect the source table to be stopped before the action is performed. Before you run the `migrate` procedure:

- Stop all writers that interact with the source table.
- Modify readers and writers that don't natively support Iceberg to enable Iceberg support.

For example:

- Athena continues to work without modification.
- Spark requires:
 - Iceberg Java Archive (JAR) files to be included in the classpath (see the [Working with Iceberg in Amazon EMR](#) and [Working with Iceberg in AWS Glue](#) sections earlier in this guide).
 - The following Spark session catalog configurations (using `SparkSessionCatalog` to add Iceberg support while maintaining built-in catalog functionalities for non-Iceberg tables):
 - `"spark.sql.extensions":"org.apache.iceberg.spark.extensions.IcebergSparkSes"`
 - `"spark.sql.catalog.spark_catalog":"org.apache.iceberg.spark.SparkSessionCat"`
 - `"spark.sql.catalog.spark_catalog.type":"glue"`
 - `"spark.hadoop.hive.metastore.client.factory.class":"com.amazonaws.glue.cata"`

After you run the procedure, you can restart your writers with their new Iceberg configuration.

Currently, the `migrate` procedure isn't compatible with the AWS Glue Data Catalog, because the Data Catalog doesn't support the `RENAME` operation. Therefore, we recommend that you use this

procedure only when you're working with Hive Metastore. If you're using the Data Catalog, see the [next section](#) for an alternative approach.

You can run the `migrate` procedure across all Amazon EMR deployment models (Amazon EMR on EC2, Amazon EMR on EKS, EMR Serverless) and AWS Glue, but it requires a configured connection to Hive Metastore. Amazon EMR on EC2 is the recommended choice because it provides a built-in Hive Metastore configuration, which minimizes setup complexity.

To test in-place migration with the `migrate` Spark procedure from an Amazon EMR on EC2 cluster that's configured with Hive Metastore, follow these steps:

1. Launch a Spark application and configure the Spark session to use the Iceberg Hive catalog implementation. For example, if you're using the `pyspark` CLI:

```
pyspark --conf
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions
--conf spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog
--conf spark.sql.catalog.spark_catalog.type=hive
```

2. Create a `products` table in Hive Metastore. This is the source table, which already exists in a typical migration.
 - a. Create the `products` external Hive table in Hive Metastore to point to the existing data in Amazon S3:

```
spark.sql(f"""
CREATE EXTERNAL TABLE products (
  product_id INT,
  product_name STRING
)
PARTITIONED BY (category STRING)
STORED AS PARQUET
LOCATION 's3://amzn-s3-demo-bucket/products/';
""")
```

- b. Add the existing partitions by using the `MSCK REPAIR TABLE` command:

```
spark.sql(f"""
MSCK REPAIR TABLE products
""")
```

c. Confirm that the table contains data by running a SELECT query:

```
spark.sql(f"""
SELECT * FROM products
""")
).show(truncate=False)
```

Sample output:

```
>>> spark.sql(f"""
... SELECT * FROM products
... """)
... ).show(truncate=False)
+-----+-----+-----+
|product_id|product_name|category|
+-----+-----+-----+
|1001      |Smartphone  |electronics|
|1002      |Laptop      |electronics|
|2001      |T-Shirt     |clothing   |
|2002      |Jeans       |clothing   |
+-----+-----+-----+
```

3. Use the Iceberg migrate procedure:

```
df_res=spark.sql(f"""
CALL system.migrate(
table => 'default.products'
)
""")

df_res.show()
```

The output dataframe contains the `migrated_files_count` (the numbers of files that were added to the Iceberg table):

```
>>> df_res.show()
+-----+
|migrated_files_count|
+-----+
|                    2|
+-----+
```

4. Confirm that the backup table was created:

```
spark.sql("show tables").show()
```

Sample output:

```
>>> spark.sql("show tables").show()
+-----+-----+-----+
| namespace |      tableName | isTemporary |
+-----+-----+-----+
| default  |      products  |      false  |
| default  | products_backup_ |      false  |
+-----+-----+-----+
```

5. Validate the operation by querying the Iceberg table:

```
spark.sql(f"""
SELECT * FROM products
""")
).show(truncate=False)
```

Notes

- After you run the procedure, all current processes that query or write to the source table will be impacted if they aren't properly configured with Iceberg support. Therefore, we recommend that you follow these steps:
 1. Stop all processes by using the source table before migration.
 2. Perform the migration.
 3. Reactivate the processes by using the proper Iceberg settings.
- If data file modifications occur during the migration process (new files are added or files are removed), the generated table will get out of sync. For synchronization options, see [Keeping Iceberg tables in sync after in-place migration](#) later in this section.

Replicating the table migration procedure in AWS Glue Data Catalog

You can replicate the migrate procedure's outcome in AWS Glue Data Catalog (backing up the original table and replacing it with an Iceberg table) by following these steps:

1. Use the snapshot procedure to create a new Iceberg table that points to the original table's data files.
2. Back up the original table metadata in the Data Catalog:
 - a. Use the [GetTable](#) API to retrieve the source table definition.
 - b. Use the [GetPartitions](#) API to retrieve the source table partition definition.
 - c. Use the [CreateTable](#) API to create a backup table in the Data Catalog.
 - d. Use the [CreatePartition](#) or [BatchCreatePartition](#) API to register partitions to the backup table in the Data Catalog.
3. Change the `gc.enabled` Iceberg table property to `false` to enable full table operations.
4. Drop the original table.
5. Locate the Iceberg table metadata JSON file in the metadata folder of the table's root location.
6. Register the new table in the Data Catalog by using the [register_table](#) procedure with the original table name and the location of the `metadata.json` file that was created by the snapshot procedure:

```
spark.sql(f"""
CALL system.register_table(
  table => 'mydb.products',
  metadata_file => '{iceberg_metadata_file}'
)
""")
).show(truncate=False)
```

Keeping Iceberg tables in sync after in-place migration

The `add_files` procedure provides a flexible way to incorporate existing data into Iceberg tables. Specifically, it registers existing data files (such as Parquet files) by referencing their absolute paths in Iceberg's metadata layer. By default, the procedure adds files from all table partitions to an Iceberg table, but you can selectively add files from specific partitions. This selective approach is particularly useful in several scenarios:

- When new partitions are added to the source table after initial migration.
- When data files are added to or removed from existing partitions after initial migration. However, re-adding modified partitions requires partition deletion first. More information about this is provided later in this section.

Here are some considerations for using the `add_file` procedure after in-place migration (snapshot or migrate) has been performed, to keep the new Iceberg table in sync with the source data files:

- When new data is added to new partitions in the source table, use the `add_files` procedure with the `partition_filter` option to selectively incorporate these additions into the Iceberg table:

```
spark.sql(f"""
CALL system.add_files(
source_table => 'mydb.products',
table => 'mydb.products_iceberg',
partition_filter => map('category', 'electronics')
).show(truncate=False)
```

or:

```
spark.sql(f"""
CALL system.add_files(
source_table => '`parquet`.`s3://amzn-s3-demo-bucket/products/`',
table => 'mydb.products_iceberg',
partition_filter => map('category', 'electronics')
).show(truncate=False)
```

- The `add_files` procedure scans for files either in the entire source table or in specific partitions when you specify the `partition_filter` option, and attempts to add all files it finds to the Iceberg table. By default, the `check_duplicate_files` procedure property is set to `true`, which prevents the procedure from running if files already exist in the Iceberg table. This is important because there is no built-in option to skip previously added files, and disabling `check_duplicate_files` will cause files to be added twice, creating duplicates. When new files are added to the source table, follow these steps:
 1. For new partitions, use `add_files` with a `partition_filter` to import only files from the new partition.
 2. For existing partitions, first delete the partition from the Iceberg table, and then re-run `add_files` for that partition, specifying the `partition_filter`. For example:

```
# We initially perform in-place migration with snapshot
spark.sql(f"""
CALL system.snapshot(
```

```
source_table => 'mydb.products',
table => 'mydb.products_iceberg',
location => 's3://amzn-s3-demo-bucket/products_iceberg/'
)
"""
).show(truncate=False)

# Then on the source table, some new files were generated under the
category='electronics' partition. Example:
spark.sql("""
INSERT INTO mydb.products
VALUES (1003, 'Tablet', 'electronics')
""")

# We delete the modified partition from the Iceberg table. Note this is a metadata
operation only
spark.sql("""
DELETE FROM mydb.products_iceberg WHERE category = 'electronics'
""")

# We add_files from the modified partition
spark.sql("""
CALL system.add_files(
  source_table => 'mydb.products',
  table => 'mydb.products_iceberg',
  partition_filter => map('category', 'electronics')
)
""").show(truncate=False)
```

Note

Every `add_files` operation generates a new Iceberg table snapshot with appended data.

Choosing the right in-place migration strategy

To choose the best in-place migration strategy, consider the questions in the following table.

Question	Recommendation	Explanation
<p>Do you want to quickly migrate without rewriting data while keeping both Hive and Iceberg tables accessible for testing or gradual transition?</p>	<p>snapshot procedure followed by <code>add_files</code> procedure</p>	<p>Use the snapshot procedure to create a new Iceberg table by cloning the schema and referencing data files, without modifying the source table. Use the <code>add_files</code> procedure to incorporate partitions that were added or modified after migration, noting that re-adding modified partitions requires partition deletion first.</p>
<p>Are you using Hive Metastore and do you want to replace your Hive table with an Iceberg table immediately, without rewriting the data?</p>	<p><code>migrate</code> procedure followed by <code>add_files</code> procedure</p>	<p>Use the <code>migrate</code> procedure to create an Iceberg table, back up the source table, and replace the original table with the Iceberg version.</p> <p>Note: This option is compatible with Hive Metastore but not with AWS Glue Data Catalog.</p> <p>Use the <code>add_files</code> procedure to incorporate partitions that were added or modified after migration, noting that re-adding modified partitions requires partition deletion first.</p>
<p>Are you using AWS Glue Data Catalog and do you want to replace your Hive table with</p>	<p>Adaptation of the <code>migrate</code> procedure, followed by <code>add_files</code> procedure</p>	<p>Replicate <code>migrate</code> procedure behavior:</p> <ol style="list-style-type: none"> 1. Use snapshot to create an Iceberg table.

Question	Recommendation	Explanation
<p>an Iceberg table immediately, without rewriting the data?</p>		<ol style="list-style-type: none"> 2. Back up the original table metadata by using AWS Glue APIs. 3. Enable <code>gc.enabled</code> on Iceberg table properties. 4. Drop the original table. 5. Use <code>register_table</code> to create a new table entry with the original name. <p>Note: This option requires manual handling of AWS Glue API calls for metadata backup.</p> <p>Use the <code>add_files</code> procedure to incorporate partitions that were added or modified after migration, noting that re-adding modified partitions requires partition deletion first.</p>

Full data migration

Full data migration recreates the data files as well as the metadata. This approach takes longer and requires additional computing resources compared with in-place migration. However, full data migration offers significant opportunities to improve table quality and optimize data storage and access patterns.

During full data migration, you can perform several beneficial operations, such as data validation to ensure integrity and correctness, schema modifications to better meet current requirements, and partition strategy adjustments for improved query performance. You can also re-sort data

to optimize common access patterns, implement Iceberg hidden partitioning for enhanced query efficiency, and perform file format conversion (for example, from CSV to Parquet) if desired.

These capabilities make full data migration ideal for transitioning to Iceberg format and for comprehensively refining and optimizing your data storage strategy. Although full data migration requires more time and resources up front, the resulting improvements in data quality, organization, and query performance can provide long-term benefits. To implement full data migration, use one of the following options:

- Use the `CREATE TABLE . . . AS SELECT (CTAS)` statement in Spark (on Amazon EMR or AWS Glue) or in Athena. You can set the partition specification and table properties for the new Iceberg table by using the `PARTITIONED BY` and `TBLPROPERTIES` clauses. You can change the schema and partitioning for the new table according to your needs instead of inheriting them from the source table.
- Read from the source table and write the data as a new Iceberg table by using Spark on Amazon EMR or AWS Glue. For more information, see [Creating a table](#) in the Iceberg documentation.

Choosing a migration strategy

When transitioning to Iceberg format, the choice between in-place and full migration is crucial. To determine the most suitable approach for your specific needs, consider the following questions and recommendations:

Question	Recommendation
What is the data file format (for example, CSV or Apache Parquet)?	<ul style="list-style-type: none"> • Consider in-place migration if your table file format is Parquet, ORC, or Avro. • For other formats such as CSV, JSON, and so on, use full data migration.
Do you want to update or consolidate the table schema?	<ul style="list-style-type: none"> • If you want to evolve the table schema by using Iceberg native capabilities, consider in-place migration. For example, you can rename columns after the migration. (The schema can be changed in the Iceberg metadata layer.)

Question	Recommendation
	<ul style="list-style-type: none"> If you want to remove entire columns because they are no longer needed, we recommend that you use full data migration.
<p>Would the table benefit from changing the partition strategy?</p>	<ul style="list-style-type: none"> If Iceberg's partitioning approach meets your requirements (for example, new data is stored by using the new partition layout while existing partitions remain as is), consider in-place migration. If you want to use hidden partitions in your table, consider full data migration. For more information about hidden partitions, see the Best practices section.
<p>Would the table benefit from adding or changing the sort order strategy?</p>	<ul style="list-style-type: none"> Adding or changing the sort order of your data requires rewriting the dataset. In this case, consider using full data migration. For large tables where it's prohibitively expensive to rewrite all the table partitions, consider using in-place migration and run compaction (with sorting enabled) for the most frequently accessed partitions.
<p>Does the table have many small files?</p>	<ul style="list-style-type: none"> Merging small files into larger files requires rewriting the dataset. In this case, consider using full data migration. For large tables where it's prohibitively expensive to rewrite all the table partitions, consider using in-place migration and run compaction (with sorting enabled) for the most frequently accessed partitions.

Migration options summary

This table summarizes the main characteristics and considerations for each migration option.

Feature	In-place migration <u>snapshot</u>	In-place migration <u>migrate</u>	Full data migration <u>CTAS or (CREATE TABLE + INSERT)</u>
<p>Data layout improvements as part of the migration process</p>			
<ul style="list-style-type: none"> • Re-sort data 	⊗No	⊗No	⊙Yes
<ul style="list-style-type: none"> • Changing partitioning (for example to use Iceberg hidden partitioning) 	⊗No	⊗No	⊙Yes
<ul style="list-style-type: none"> • Changing table schema 	⊗No	⊗No	⊙Yes
<ul style="list-style-type: none"> • Optimizing file size 	⊗No	⊗No	⊙Yes

Feature	In-place migration <u>snapshot</u>	In-place migration <u>migrate</u>	Full data migration <u>CTAS or (CREATE TABLE + INSERT)</u>
<ul style="list-style-type: none"> Validate the schema of existing data before adding the data 	⊗No	⊗No	⊙Yes
Support file formats	Parquet, Avro, ORC	Parquet, Avro, ORC	Parquet, Avro, ORC, JSON, CSV
Source table replaced by an Iceberg table	⊗No (creates a new table, but with additional steps you can replace the source table)	⊙Yes (creates a backup table and substitutes the source table with an Iceberg table)	⊗No (creates a new table)
Source table impact			

Feature	In-place migration <u>snapshot</u>	In-place migration <u>migrate</u>	Full data migration <u>CTAS or (CREATE TABLE + INSERT)</u>
<ul style="list-style-type: none"> File deletion operations on Iceberg table (explicit snapshot operations, dropping a table with purge) 	Corrupts source table	Corrupts backup table	Safe, source unaffected
<p>Iceberg table impact</p>			
<ul style="list-style-type: none"> Impact if source table files are removed 	Corrupts Iceberg table	Corrupts Iceberg table	No impact on Iceberg table

Feature	In-place migration <u>snapshot</u>	In-place migration <u>migrate</u>	Full data migration <u>CTAS or (CREATE TABLE + INSERT)</u>
<ul style="list-style-type: none"> Impact if new files are added on source table location 	Not visible on new table (need to incorporate partition with add_files)	Not visible on new table (need to incorporate partition with add_files)	Not visible on new table (need to INSERT INTO the new table)
Cost	Low	Low	Higher (full data rewrite)
Migration speed	Fast	Fast	Slower
Can be used to migrate to Amazon S3 Tables	⊗No	⊗No	⊙Yes
Requires manual DDL	⊗No (schema and partitions are copied from source table)	⊗No (schema and partitions are copied from source table)	If using CTAS, requires only specifying the partitioning

Feature	In-place migration <u>snapshot</u>	In-place migration <u>migrate</u>	Full data migration <u>CTAS or (CREATE TABLE + INSERT)</u>
Best use	Quick migration without rewriting data, allowing side-by-side use of Hive and Iceberg for testing or gradual transition.	Replacing a Hive table in place without rewriting data, when an immediate switchover is acceptable.	Full Iceberg optimization with data rewrite. Ideal when redesigning partitions or schema, or improving layout and performance. Always recommended if possible.

Best practices for optimizing Apache Iceberg workloads

Iceberg is a table format that's designed to simplify data lake management and enhance workload performance. Different use cases might prioritize different aspects such as cost, read performance, write performance, or data retention, so Iceberg offers configuration options to manage these trade-offs. This section provides insights for optimizing and fine-tuning your Iceberg workloads to meet your requirements.

Topics

- [General best practices](#)
- [Optimizing read performance](#)
- [Optimizing write performance](#)
- [Optimizing storage](#)
- [Maintaining tables by using compaction](#)
- [Using Iceberg workloads in Amazon S3](#)

General best practices

Regardless of your use case, when you use Apache Iceberg on AWS, we recommend that you follow these general best practices.

- **Use Iceberg format version 2.**

Athena uses Iceberg format version 2 by default.

When you use Spark on Amazon EMR or AWS Glue to create Iceberg tables, specify the format version as described in the [Iceberg documentation](#).

- **Use the AWS Glue Data Catalog as your data catalog.**

Athena uses the AWS Glue Data Catalog by default.

When you use Spark on Amazon EMR or AWS Glue to work with Iceberg, add the following configuration to your Spark session to use the AWS Glue Data Catalog. For more information, see the section [Spark configurations for Iceberg in AWS Glue](#) earlier in this guide.

```
"spark.sql.catalog.<your_catalog_name>.type": "glue"
```

- **Use the AWS Glue Data Catalog as lock manager.**

Athena uses the AWS Glue Data Catalog as lock manager by default for Iceberg tables.

When you use Spark on Amazon EMR or AWS Glue to work with Iceberg, make sure to configure your Spark session configuration to use the AWS Glue Data Catalog as lock manager. For more information, see [Optimistic Locking](#) in the Iceberg documentation.

- **Use Zstandard (ZSTD) compression.**

The default compression codec of Iceberg is gzip, which can be modified by using the table property `write.<file_type>.compression-codec`. Athena already uses ZSTD as the default compression codec for Iceberg tables.

In general, we recommend using the ZSTD compression codec because it strikes a balance between GZIP and Snappy, and offers good read/write performance without compromising the compression ratio. Additionally, compression levels can be adjusted to suit your needs. For more information, see [ZSTD compression levels in Athena](#) in the Athena documentation.

Snappy might provide the best overall read and write performance but has a lower compression ratio than GZIP and ZSTD. If you prioritize performance—even if it means storing larger data volumes in Amazon S3—Snappy might be the optimal choice.

Optimizing read performance

This section discusses table properties that you can tune to optimize read performance, independent of the engine.

Partitioning

As with Hive tables, Iceberg uses partitions as the primary layer of indexing to avoid reading unnecessary metadata files and data files. Column statistics are also taken into consideration as a secondary layer of indexing to further improve query planning, which leads to better overall execution time.

Partition your data

To reduce the amount of data that's scanned when querying Iceberg tables, choose a balanced partition strategy that aligns with your expected read patterns:

- Identify columns that are frequently used in queries. These are ideal partitioning candidates. For example, if you typically query data from a particular day, a natural example of a partition column would be a date column.
- Choose a low cardinality partition column to avoid creating an excessive number of partitions. Too many partitions can increase the number of files in the table, which can negatively impact query performance. As a rule of thumb, "too many partitions" can be defined as a scenario where the data size in the majority of partitions is less than 2-5 times the value set by `target-file-size-bytes`.

Note

If you typically query by using filters on a high cardinality column (for example, an `id` column that can have thousands of values), use Iceberg's hidden partitioning feature with bucket transforms, as explained in the next section.

Use hidden partitioning

If your queries commonly filter on a derivative of a table column, use hidden partitions instead of explicitly creating new columns to work as partitions. For more information about this feature, see the [Iceberg documentation](#).

For example, in a dataset that has a timestamp column (for example, `2023-01-01 09:00:00`), instead of creating a new column with the parsed date (for example, `2023-01-01`), use partition transforms to extract the date part from the timestamp and create these partitions on the fly.

The most common use cases for hidden partitioning are:

- **Partitioning on date or time**, when the data has a timestamp column. Iceberg offers multiple transforms to extract the date or time parts of a timestamp.
- **Partitioning on a hash function of a column**, when the partitioning column has high cardinality and would result in too many partitions. Iceberg's bucket transform groups multiple partition values together into fewer, hidden (bucket) partitions by using hash functions on the partitioning column.

See [partition transforms](#) in the Iceberg documentation for an overview of all available partition transforms.

Columns that are used for hidden partitioning can become part of query predicates through the use of regular SQL functions such as `year()` and `month()`. Predicates can also be combined with operators such as `BETWEEN` and `AND`.

Note

Iceberg cannot perform partition pruning for functions that yield a different data type; for example, `substring(event_time, 1, 10) = '2022-01-01'`.

Use partition evolution

Use [Iceberg's partition evolution](#) when the existing partition strategy isn't optimal. For example, if you choose hourly partitions that turn out to be too small (just a few megabytes each), consider shifting to daily or monthly partitions.

You can use this approach when the best partition strategy for a table is initially unclear, and you want to refine your partitioning strategy as you gain more insights. Another effective use of partition evolution is when data volumes change and the current partitioning strategy becomes less effective over time.

For instructions on how to evolve partitions, see [ALTER TABLE SQL extensions](#) in the Iceberg documentation.

Tuning file sizes

Optimizing query performance involves minimizing the number of small files in your tables. For good query performance, we generally recommend keeping Parquet and ORC files larger than 100 MB.

File size also impacts query planning for Iceberg tables. As the number of files in a table increases, so does the size of the metadata files. Larger metadata files can result in slower query planning. Therefore, when the table size grows, increase the file size to alleviate the exponential expansion of metadata.

Use the following best practices to create properly sized files in Iceberg tables.

Set target file and row group size

Iceberg offers the following key configuration parameters for tuning the data file layout. We recommend that you use these parameters to set the target file size and row group or stripe size.

Parameter	Default value	Comment
<code>write.target-file-size-bytes</code>	512 MB	This parameter specifies the maximum file size that Iceberg will create. However, certain files might be written with a smaller size than this limit.
<code>write.parquet.row-group-size-bytes</code>	128 MB	Both Parquet and ORC store data in chunks so that engines can avoid reading the entire file for some operations.
<code>write.orc.stripe-size-bytes</code>	64 MB	
<code>write.distribution-mode</code>	None, for Iceberg version 1.1 and lower Hash, starting with Iceberg version 1.2	Iceberg requests Spark to sort data between its tasks before writing to storage.

- Based on your expected table size, follow these general guidelines:
 - **Small tables** (up to few gigabytes) – Reduce the target file size to 128 MB. Also reduce the row group or stripe size (for example, to 8 or 16 MB).
 - **Medium to large tables** (from a few gigabytes to hundreds of gigabytes) – The default values are a good starting point for these tables. If your queries are very selective, adjust the row group or stripe size (for example, to 16 MB).
 - **Very large tables** (hundreds of gigabytes or terabytes) – Increase the target file size to 1024 MB or more, and consider increasing the row group or stripe size if your queries usually pull large sets of data.
- To ensure that Spark applications that write to Iceberg tables create appropriately sized files, set the `write.distribution-mode` property to either `hash` or `range`. For a detailed

explanation of the difference between these modes, see [Writing Distribution Modes](#) in the Iceberg documentation.

These are general guidelines. We recommend that you run tests to identify the most suitable values for your specific tables and workloads.

Run regular compaction

The configurations in the previous table set a maximum file size that write tasks can create, but do not guarantee that files will have that size. To ensure proper file sizes, run compaction regularly to combine small files into larger files. For detailed guidance on running compaction, see [Iceberg compaction](#) later in this guide.

Optimize column statistics

Iceberg uses column statistics to perform file pruning, which improves query performance by reducing the amount of data that's scanned by queries. To benefit from column statistics, make sure that Iceberg collects statistics for all columns that are frequently used in query filters.

By default, Iceberg collects statistics only for the [first 100 columns in each table](#), as defined by the table property `write.metadata.metrics.max-inferred-column-defaults`. If your table has more than 100 columns and your queries frequently reference columns outside of the first 100 columns (for example, you might have queries that filter on column 132), make sure that Iceberg collects statistics on those columns. There are two options to achieve this:

- When you create the Iceberg table, reorder columns so that the columns you need for queries fall within the column range set by `write.metadata.metrics.max-inferred-column-defaults` (default is 100).

Note: If you don't need statistics on 100 columns, you can adjust the `write.metadata.metrics.max-inferred-column-defaults` configuration to a desired value (for example, 20) and reorder the columns so that the columns you need to read and write queries fall within the first 20 columns on the left side of the dataset.

- If you use only a few columns in query filters, you can disable the overall property for metrics collection and selectively choose individual columns to collect statistics for, as shown in this example:

```
.tableProperty("write.metadata.metrics.default", "none")
```

```
.tableProperty("write.metadata.metrics.column.my_col_a", "full")  
.tableProperty("write.metadata.metrics.column.my_col_b", "full")
```

Note: Column statistics are most effective when data is sorted on those columns. For more information, see the [Set the sort order](#) section later in this guide.

Choose the right update strategy

Use a copy-on-write strategy to optimize read performance, when slower write operations are acceptable for your use case. This is the default strategy used by Iceberg.

Copy-on-write results in better read performance, because files are directly written to storage in a read-optimized fashion. However, compared with merge-on-read, each write operation takes longer and consumes more compute resources. This presents a classic trade-off between read and write latency. Typically, copy-on-write is ideal for use cases where most updates are collocated in the same table partitions (for example, for daily batch loads).

Copy-on-write configurations (`write.update.mode`, `write.delete.mode`, and `write.merge.mode`) can be set at the table level or independently on the application side.

Use ZSTD compression

You can modify the compression codec used by Iceberg by using the table property `write.<file_type>.compression-codec`. We recommend that you use the ZSTD compression codec to improve overall performance on tables.

By default, Iceberg versions 1.3 and earlier use GZIP compression, which provides slower read/write performance compared with ZSTD.

Note: Some engines might use different default values. This is the case for [Iceberg tables that are created with Athena](#) or Amazon EMR version 7.x.

Set the sort order

To improve read performance on Iceberg tables, we recommend that you sort your table based on one or more columns that are frequently used in query filters. Sorting, combined with Iceberg's column statistics, can make file pruning significantly more efficient, which results in faster read operations. Sorting also reduces the number of Amazon S3 requests for queries that use the sort columns in query filters.

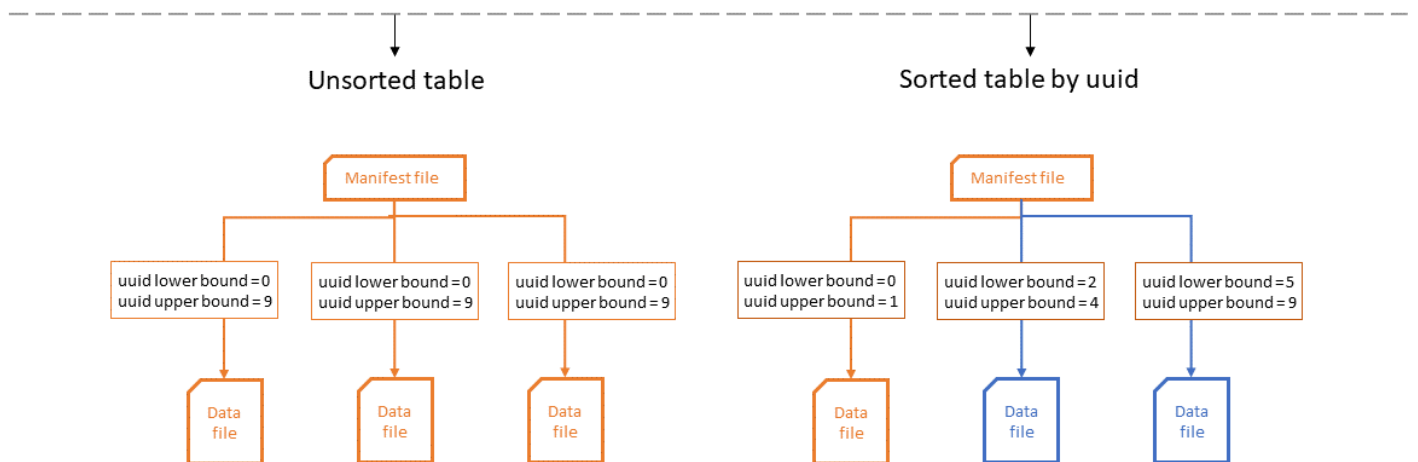
You can set a hierarchical sort order at the table level by running a data definition language (DDL) statement with Spark. For available options, see the [Iceberg documentation](#). After you set the sort order, writers will apply this sorting to subsequent data write operations in the Iceberg table.

For example, in tables that are partitioned by date (yyyy-mm-dd) where most of the queries filter by uuid, you can use the DDL option `Write Distributed By Partition Locally Ordered` to make sure that Spark writes files with non-overlapping ranges.

The following diagram illustrates how the efficiency of column statistics improves when tables are sorted. In the example, the sorted table needs to open only a single file, and maximally benefits from Iceberg's partition and file. In the unsorted table, any uuid can potentially exist in any data file, so the query has to open all data files.

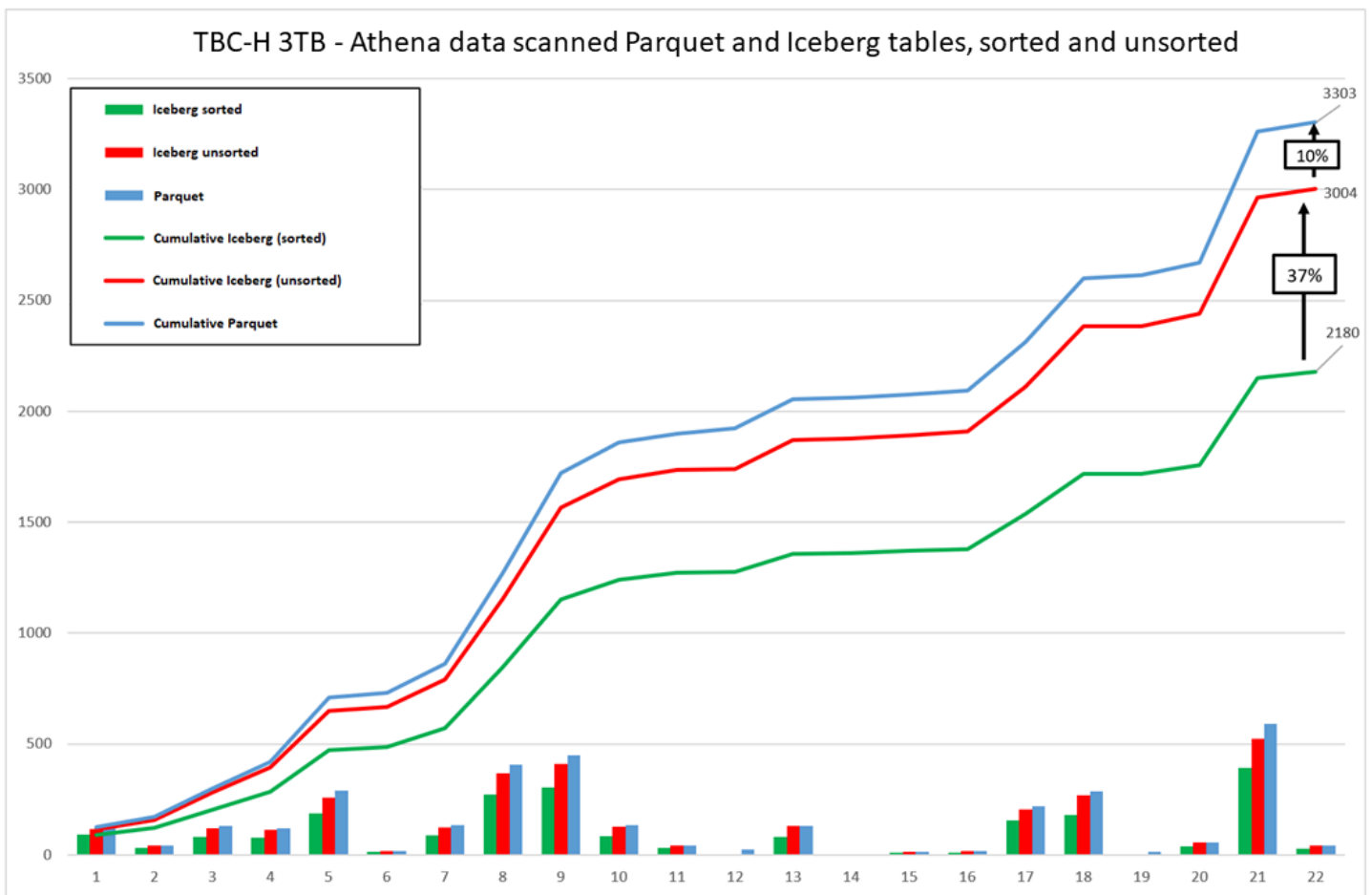
Query example:

```
SELECT * FROM Table
WHERE date > 2022-02-05 AND date < 2022-02-10 AND uuid = 1
```



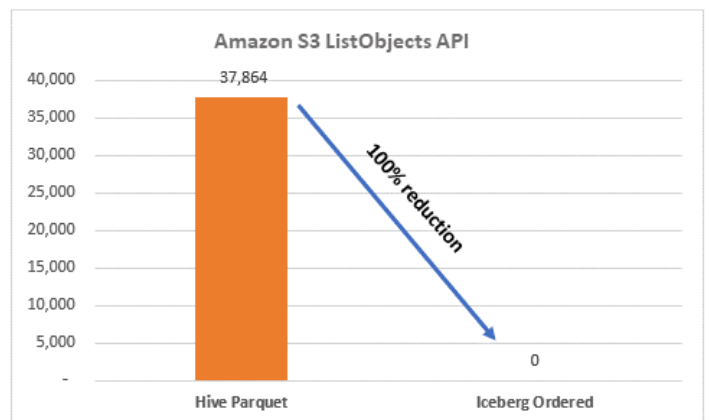
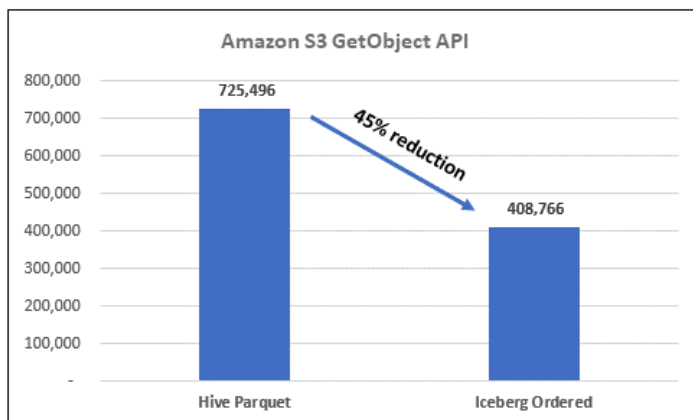
Changing the sort order doesn't affect existing data files. You can use Iceberg compaction to apply the sort order on those.

Using Iceberg sorted tables might decrease costs for your workload, as illustrated in the following graph.



These graphs summarize the results of running the TPC-H benchmark for Hive (Parquet) tables compared with Iceberg sorted tables. However, the results might be different for other datasets or workloads.

TPC-H 3TB - 22 queries



Optimizing write performance

This section discusses table properties that you can tune to optimize write performance on Iceberg tables, independent of the engine.

Set the table distribution mode

Iceberg offers multiple write distribution modes that define how write data is distributed across Spark tasks. For an overview of the available modes, see [Writing Distribution Modes](#) in the Iceberg documentation.

For use cases that prioritize write speed, especially in streaming workloads, set `write.distribution-mode` to `none`. This ensures that Iceberg doesn't request additional Spark shuffling and that data is written as it becomes available in Spark tasks. This mode is particularly suitable for Spark Structured Streaming applications.

Note

Setting the write distribution mode to `none` tends to produce numerous small files, which degrades read performance. We recommend regular compaction to consolidate these small files into properly sized files for query performance.

Choose the right update strategy

Use a merge-on-read strategy to optimize write performance, when slower read operations on the latest data are acceptable for your use case.

When you use merge-on-read, Iceberg writes updates and deletes to storage as separate small files. When the table is read, the reader has to merge these changes with the base files to return the latest view of the data. This results in a performance penalty for read operations, but speeds up the writing of updates and deletes. Typically, merge-on-read is ideal for streaming workloads with updates or jobs with few updates that are spread across many table partitions.

You can set merge-on-read configurations (`write.update.mode`, `write.delete.mode`, and `write.merge.mode`) at the table level or independently on the application side.

Using merge-on-read requires running regular compaction to prevent read performance from degrading over time. Compaction reconciles updates and deletes with existing data files to create

a new set of data files, thereby eliminating the performance penalty incurred on the read side. By default, Iceberg's compaction doesn't merge delete files unless you change the default of the `delete-file-threshold` property to a smaller value (see the [Iceberg documentation](#)). To learn more about compaction, see the section [Iceberg compaction](#) later in this guide.

Choose the right file format

Iceberg supports writing data in Parquet, ORC, and Avro formats. Parquet is the default format. Parquet and ORC are columnar formats that offer superior read performance but are generally slower to write. This represents the typical trade-off between read and write performance.

If write speed is important for your use case, such as in streaming workloads, consider writing in Avro format by setting `write-format` to Avro in the writer's options. Because Avro is a row-based format, it provides faster write times at the cost of slower read performance.

To improve read performance, run regular compaction to merge and transform small Avro files into larger Parquet files. The outcome of the compaction process is governed by the `write.format.default` table setting. The default format for Iceberg is Parquet, so if you write in Avro and then run compaction, Iceberg will transform the Avro files into Parquet files. Here's an example:

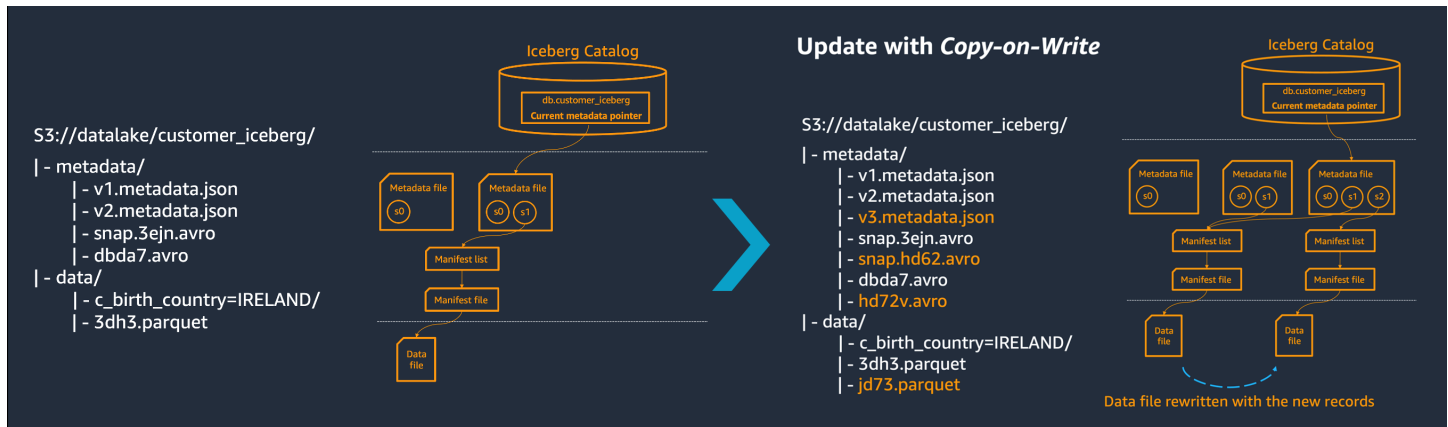
```
spark.sql(f"""
    CREATE TABLE IF NOT EXISTS glue_catalog.{DB_NAME}.{TABLE_NAME} (
        Col_1 float,
        <<<...other columns...>>
        ts timestamp)
    USING iceberg
    PARTITIONED BY (days(ts))
    OPTIONS (
        'format-version'='2',
        write.format.default='parquet')
    """)

query = df \
    .writeStream \
    .format("iceberg") \
    .option("write-format", "avro") \
    .outputMode("append") \
    .trigger(processingTime='60 seconds') \
    .option("path", f"glue_catalog.{DB_NAME}.{TABLE_NAME}") \
    .option("checkpointLocation", f"s3://{BUCKET_NAME}/checkpoints/iceberg/")
```

```
.start()
```

Optimizing storage

Updating or deleting data in an Iceberg table increases the number of copies of your data, as illustrated in the following diagram. The same is true for running compaction: It increases the number of data copies in Amazon S3. That's because Iceberg treats the files underlying all tables as immutable.



Follow the best practices in this section to manage storage costs.

Enable S3 Intelligent-Tiering

Use the [Amazon S3 Intelligent-Tiering](#) storage class to automatically move data to the most cost-effective access tier when access patterns change. This option has no operational overhead or impact on performance.

Note: Don't use the optional tiers (such as Archive Access and Deep Archive Access) in S3 Intelligent-Tiering with Iceberg tables. To archive data, see the guidelines in the next section.

You can also use [Amazon S3 Lifecycle rules](#) to set your own rules for moving objects to another Amazon S3 storage class, such as S3 Standard-IA or S3 One Zone-IA (see [Supported transitions and related constraints](#) in the Amazon S3 documentation).

Archive or delete historic snapshots

For every committed transaction (insert, update, merge into, compaction) to an Iceberg table, a new version or snapshot of the table is created. Over time, the number of versions and the number of metadata files in Amazon S3 accumulate.

Keeping snapshots of a table is required for features such as snapshot isolation, table rollback, and time travel queries. However, storage costs grow with the number of versions that you retain.

The following table describes the design patterns you can implement to manage costs based on your data retention requirements.

Design pattern	Solution	Use cases
Delete old snapshots	<ul style="list-style-type: none"> Use the VACUUM statement in Athena to remove old snapshots. This operation doesn't incur any compute cost. Alternatively, you can use Spark on Amazon EMR or AWS Glue to remove snapshots. For more information, see expire_snapshots in the Iceberg documentation. 	<p>This approach deletes snapshots that are no longer needed to reduce storage costs. You can configure how many snapshots should be retained or for how long, based on your data retention requirements.</p> <p>This option performs a hard delete of the snapshots. You can't roll back or time travel to expired snapshots.</p>
Set retention policies for specific snapshots	<ol style="list-style-type: none"> Use tags to mark specific snapshots and define a retention policy in Iceberg. For more information, see Historical Tags in the Iceberg documentation. <p>For example, you can retain one snapshot per month for one year by using the following SQL statement in Spark on Amazon EMR:</p>	<p>This pattern is helpful for compliance with business or legal requirements that require you to show the state of a table at a given point in the past. By placing retention policies on specific tagged snapshots, you can remove other (untagged) snapshots that were created. This way, you can meet data retention requirements without retaining every single snapshot created.</p>

```
ALTER TABLE glue_cata
log.db.table
```

Design pattern**Solution**

```
CREATE TAG 'EOM-01' AS  
OF VERSION 30 RETAIN  
365 DAYS
```

Use cases

2. Use Spark on Amazon EMR or AWS Glue to remove the remaining untagged, intermediate snapshots.

Design pattern

Archive old snapshots

Solution

1. Use Amazon S3 tags to mark objects with Spark. (Amazon S3 tags are different from Iceberg tags; for more information, see the [Iceberg documentation](#).) For example:

```
spark.sql.catalog.  
my_catalog.s3.delete-enabled=false and  
\  
spark.sql.catalog.  
my_catalog.s3.delete.tags.my_key=to_archive
```

2. Use Spark on Amazon EMR or AWS Glue to [remove snapshots](#). When you use the settings in the example, this procedure tags objects and detaches them from the Iceberg table metadata instead of deleting them from Amazon S3.
3. Use S3 Life cycle rules to transition objects tagged as `to_archive` to one of the [S3 Glacier storage classes](#).
4. To query archived data:
 - [Restore the archived objects](#) (this step isn't required if objects were

Use cases

This pattern allows you to keep all table versions and snapshots at a lower cost.

You cannot time travel or roll back to archived snapshots without first restoring those versions as new tables. This is typically acceptable for audit purposes.

You can combine this approach with the previous design pattern, setting retention policies for specific snapshots.

Design pattern	Solution	Use cases
	<p>transitioned to the Amazon Glacier Instant Retrieval storage class).</p> <ul style="list-style-type: none">• Use the register_table procedure in Iceberg to register the snapshot as a table in the catalog.	
	<p>For detailed instructions, see the AWS blog post Improve operational efficiencies of Apache Iceberg tables build on Amazon S3 data lakes.</p>	

Delete orphan files

In certain situations, Iceberg applications can fail before you commit your transactions. This leaves data files in Amazon S3. Because there was no commit, these files won't be associated with any table, so you might have to clean them up asynchronously.

To handle these deletions, you can use the [VACUUM statement](#) in Amazon Athena. This statement removes snapshots and also deletes orphaned files. This is very cost-efficient, because Athena doesn't charge for the compute cost of this operation. Also, you don't have to schedule any additional operations when you use the VACUUM statement.

Alternatively, you can use Spark on Amazon EMR or AWS Glue to run the `remove_orphan_files` procedure. This operation has a compute cost and has to be scheduled independently. For more information, see the [Iceberg documentation](#).

Maintaining tables by using compaction

Iceberg includes features that enable you to carry out [table maintenance operations](#) after writing data to the table. Some maintenance operations focus on streamlining metadata files, while others

enhance how the data is clustered in the files so that query engines can efficiently locate the necessary information to respond to user requests. This section focuses on compaction-related optimizations.

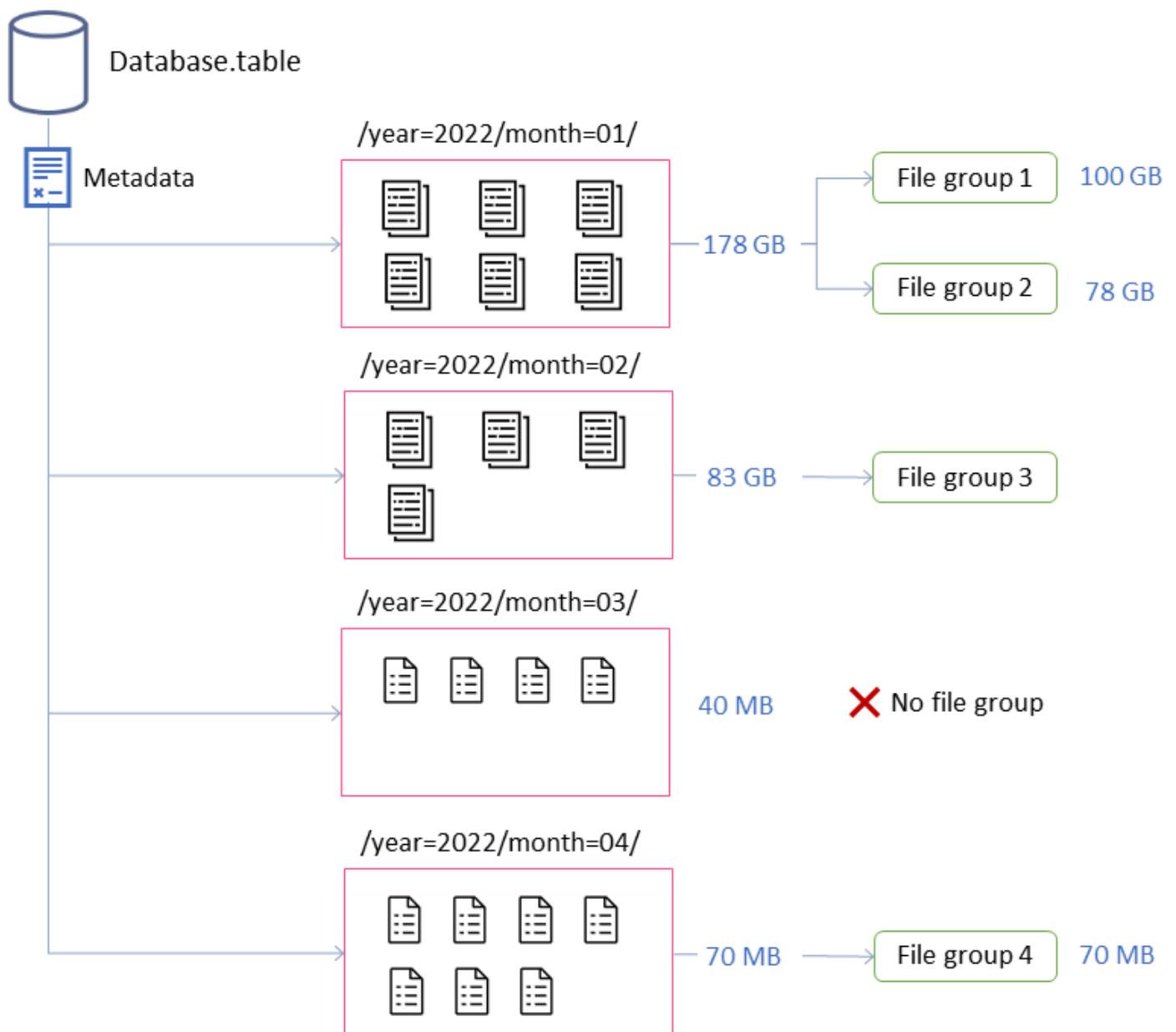
Iceberg compaction

In Iceberg, you can use compaction to perform four tasks:

- Combining small files into larger files that are generally over 100 MB in size. This technique is known as *bin packing*.
- Merging delete files with data files. Delete files are generated by updates or deletes that use the merge-on-read approach.
- (Re)sorting the data in accordance with query patterns. Data can be written without any sort order or with a sort order that is suitable for writes and updates.
- Clustering the data by using space filling curves to optimize for distinct query patterns, particularly z-order sorting.

On AWS, you can run table compaction and maintenance operations for Iceberg through Amazon Athena or by using Spark in Amazon EMR or AWS Glue.

When you run compaction by using the [rewrite_data_files](#) procedure, you can adjust several knobs to control the compaction behavior. The following diagram shows the default behavior of bin packing. Understanding bin packing compaction is key to understanding hierarchical sorting and Z-order sorting implementations, because they are extensions of the bin packing interface and operate in a similar manner. The main distinction is the additional step required for sorting or clustering the data.



In this example, the Iceberg table consists of four partitions. Each partition has a different size and different number of files. If you start a Spark application to run compaction, the application creates a total of four file groups to process. A file group is an Iceberg abstraction that represents a collection of files that will be processed by a single Spark job. That is, the Spark application that runs compaction will create four Spark jobs to process the data.

Tuning compaction behavior

The following key properties control how data files are selected for compaction:

- [MAX_FILE_GROUP_SIZE_BYTES](#) sets the data limit for a single file group (Spark job) at 100 GB by default. This property is especially important for tables without partitions or tables with partitions that span hundreds of gigabytes. By setting this limit, you can break down operations to plan work and make progress while preventing resource exhaustion on the cluster.

Note: Each file group is sorted separately. Therefore, if you want to perform a partition-level sort, you must adjust this limit to match the partition size.

- [MIN_FILE_SIZE_BYTES](#) or [MIN_FILE_SIZE_DEFAULT_RATIO](#) defaults to 75 percent of the target file size set at the table level. For example, if a table has a target size of 512 MB, any file that is smaller than 384 MB is included in the set of files that will be compacted.
- [MAX_FILE_SIZE_BYTES](#) or [MAX_FILE_SIZE_DEFAULT_RATIO](#) defaults to 180 percent of the target file size. As with the two properties that set minimum file sizes, these properties are used to identify candidate files for the compaction job.
- [MIN_INPUT_FILES](#) specifies the minimum number of files to be compacted if a table partition size is smaller than the target file size. The value of this property is used to determine whether it is worthwhile to compact the files based on the number of files (defaults to 5).
- [DELETE_FILE_THRESHOLD](#) specifies the minimum number of delete operations for a file before it's included in compaction. Unless you specify otherwise, compaction doesn't combine delete files with data files. To enable this functionality, you must set a threshold value by using this property. This threshold is specific to individual data files, so if you set it to 3, a data file will be rewritten only if there are three or more delete files that reference it.

These properties provide insight into the formation of the file groups in the previous diagram.

For example, the partition labeled `month=01` includes two file groups because it exceeds the maximum size constraint of 100 GB. In contrast, the `month=02` partition contains a single file group because it's under 100 GB. The `month=03` partition doesn't satisfy the default minimum input file requirement of five files. As a result, it won't be compacted. Lastly, although the `month=04` partition doesn't contain enough data to form a single file of the desired size, the files will be compacted because the partition includes more than five small files.

You can set these parameters for Spark running on Amazon EMR or AWS Glue. For Amazon Athena, you can manage similar properties by using the [table properties](#) that start with the prefix `optimize_`.

Running compaction with Spark on Amazon EMR or AWS Glue

This section describes how to properly size a Spark cluster to run Iceberg's compaction utility. The following example uses Amazon EMR Serverless, but you can use the same methodology in Amazon EMR on EC2 or EKS, or in AWS Glue.

You can take advantage of the correlation between file groups and Spark jobs to plan the cluster resources. To process the file groups sequentially, considering the maximum size of 100 GB per file group, you can set the following [Spark properties](#):

- `spark.dynamicAllocation.enabled = FALSE`
- `spark.executor.memory = 20 GB`
- `spark.executor.instances = 5`

If you want to speed up compaction, you can scale horizontally by increasing the number of file groups that are compacted in parallel. You can also scale Amazon EMR by using manual or dynamic scaling.

- **Manually scaling** (for example, by a factor of 4)
 - `MAX_CONCURRENT_FILE_GROUP_REWRITES = 4` (our factor)
 - `spark.executor.instances = 5` (value used in the example) x 4 (our factor) = 20
 - `spark.dynamicAllocation.enabled = FALSE`
- **Dynamic scaling**
 - `spark.dynamicAllocation.enabled = TRUE` (default, no action required)
 - [MAX_CONCURRENT_FILE_GROUP_REWRITES](#) = N (align this value with `spark.dynamicAllocation.maxExecutors`, which is 100 by default; based on the executor configurations in the example, you can set N to 20)

These are guidelines to help size the cluster. However, you should also monitor the performance of your Spark jobs to find the best settings for your workloads.

Running compaction with Amazon Athena

Athena offers an implementation of Iceberg's compaction utility as a managed feature through the [OPTIMIZE statement](#). You can use this statement to run compaction without having to evaluate the infrastructure.

This statement groups small files into larger files by using the bin packing algorithm and merges delete files with existing data files. To cluster the data by using hierarchical sorting or z-order sorting, use Spark on Amazon EMR or AWS Glue.

You can change the default behavior of the OPTIMIZE statement at table creation by passing table properties in the CREATE TABLE statement, or after table creation by using the ALTER TABLE statement. For default values, see the [Athena documentation](#).

Recommendations for running compaction

Use case

Recommendation

Running bin packing compaction based on a schedule

- Use the OPTIMIZE statement in Athena if you don't know how many small files your table contains. The Athena pricing model is based on the data scanned, so if there are no files to be compacted, there is no cost associated with these operations. To avoid encountering timeouts on Athena tables, run OPTIMIZE on a per-table-partition basis.
- Use Amazon EMR or AWS Glue with dynamic scaling when you expect large volumes of small files to be compacted.

Running bin packing compaction based on events

- Use Amazon EMR or AWS Glue with dynamic scaling when you expect large volumes of small files to be compacted.

Running compaction to sort data

- Use Amazon EMR or AWS Glue, because sorting is an expensive operation and might need to spill data to disk.

Running compaction to cluster the data using z-order sorting

- Use Amazon EMR or AWS Glue, because z-order sorting is a very expensive operation and might need to spill data to disk.

Use case

Running compaction on partitions that might be updated by other applications because of late-arriving data

Running compaction on cold partitions (data partitions that no longer receive active writes)

Recommendation

- Use Amazon EMR or AWS Glue. Enable the Iceberg [PARTIAL_PROGRESS_ENABLED](#) property. When you use this option, Iceberg splits the compaction output into multiple commits. If there is a collision (that is, if the data file is updated while compaction is running), this setting reduces the cost of retry by limiting it to the commit that includes the affected file. Otherwise, you might have to recompact all files.
- Use Amazon EMR or AWS Glue. In the `rewrite_data_files` procedure, specify a where predicate that excludes actively written partitions. This strategy prevents data conflicts between writers and compaction jobs, and leaves only metadata conflicts that Iceberg can automatically resolve.

Using Iceberg workloads in Amazon S3

This section discusses Iceberg properties that you can use to optimize Iceberg's interaction with Amazon S3.

Prevent hot partitioning (HTTP 503 errors)

Some data lake applications that run on Amazon S3 handle millions or billions of objects and process petabytes of data. This can lead to prefixes that receive a high volume of traffic, which are typically detected through HTTP 503 (service unavailable) errors. To prevent this issue, use the following Iceberg properties:

- Set `write.distribution-mode` to `hash` or `range` so that Iceberg writes large files, which results in fewer Amazon S3 requests. This is the preferred configuration and should address the majority of cases.

- If you continue to experience 503 errors due to an immense volume of data in your workloads, you can set `write.object-storage.enabled` to `true` in Iceberg. This instructs Iceberg to hash object names and distribute the load across multiple, randomized Amazon S3 prefixes.

For more information about these properties, see [Write properties](#) in the Iceberg documentation.

Use Iceberg maintenance operations to release unused data

To manage Iceberg tables, you can use the Iceberg core API, Iceberg clients (such as Spark), or managed services such as Amazon Athena. To delete old or unused files from Amazon S3, we recommend that you only use Iceberg native APIs to [remove snapshots](#), [remove old metadata files](#), and [delete orphan files](#).

Using Amazon S3 APIs through Boto3, the Amazon S3 SDK, or the AWS Command Line Interface (AWS CLI), or using any other, non-Iceberg methods to overwrite or remove Amazon S3 files for an Iceberg table leads to table corruption and query failures.

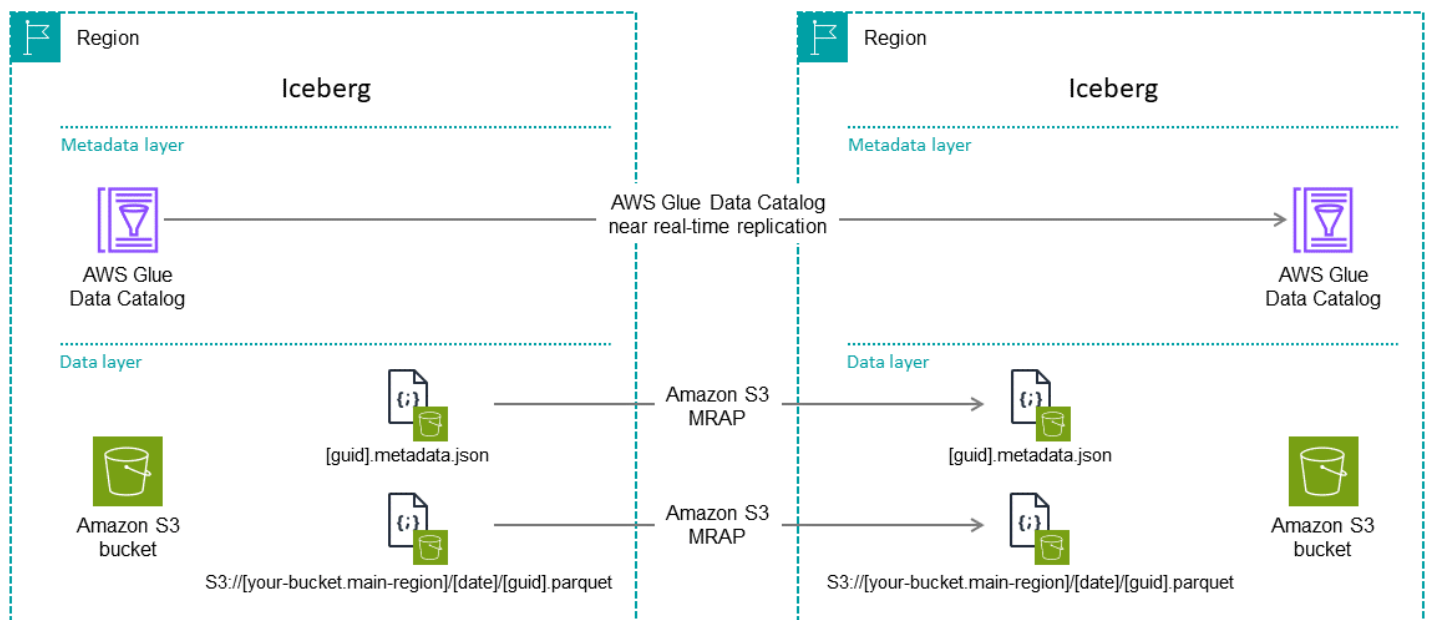
Replicate data across AWS Regions

When you store Iceberg tables in Amazon S3, you can use the built-in features in Amazon S3, such as [Cross-Region Replication \(CRR\)](#) and [Multi-Region Access Points \(MRAP\)](#), to replicate data across multiple AWS Regions. MRAP provides a global endpoint for applications to access S3 buckets that are located in multiple AWS Regions. Iceberg doesn't support relative paths, but you can use MRAP to perform Amazon S3 operations by mapping buckets to access points. MRAP also integrates seamlessly with the Amazon S3 Cross-Region Replication process, which introduces a lag of up to 15 minutes. You have to replicate both data and metadata files.

Important

Currently, Iceberg integration with MRAP works only with Apache Spark. If you need to fail over to the secondary AWS Region, you have to plan to redirect user queries to a Spark SQL environment (such as Amazon EMR) in the failover Region.

The CRR and MRAP features help you build a cross-Region replication solution for Iceberg tables, as illustrated in the following diagram.



To set up this cross-Region replication architecture:

1. Create tables by using the MRAP location. This ensures that Iceberg metadata files point to the MRAP location instead of the physical bucket location.
2. Replicate Iceberg files by using Amazon S3 MRAP. MRAP supports data replication with a service-level agreement (SLA) of 15 minutes. Iceberg prevents read operations from introducing inconsistencies during replication.
3. Make the tables available in the AWS Glue Data Catalog in the secondary Region. You can choose from two options:
 - Set up a pipeline for replicating Iceberg table metadata by using AWS Glue Data Catalog replication. This utility is available in the GitHub [Glue Catalog and Lake Formation Permissions replication](#) repository. This event-driven mechanism replicates tables in the target Region based on event logs.
 - Register the tables in the secondary Region when you need to fail over. For this option, you can use the previous utility or the Iceberg [register_table procedure](#) and point it to the latest metadata.json file.

Monitoring Apache Iceberg workloads

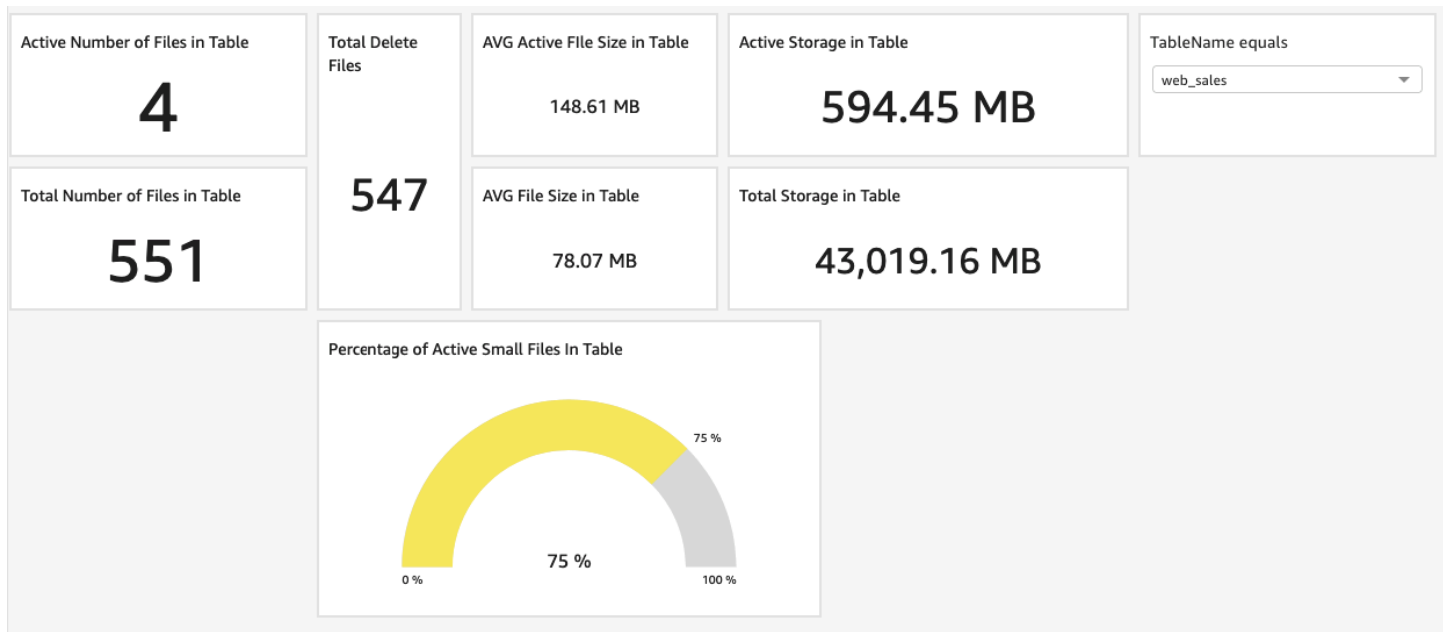
To monitor Iceberg workloads, you have two options: analyzing [metadata tables](#) or using [metrics reporters](#). Metrics reporters were introduced in Iceberg version 1.2 and are available only for REST and JDBC catalogs.

If you're using AWS Glue Data Catalog, you can gain insights into the health of your Iceberg tables by setting up monitoring on top of the metadata tables that Iceberg exposes.

Monitoring is crucial for performance management and troubleshooting. For example, when a partition in an Iceberg table reaches a certain percentage of small files, your workload can start a compaction job to consolidate the files into larger ones. This prevents queries from slowing down beyond an acceptable level.

Table-level monitoring

The following screen shows a table monitoring dashboard that was created in Amazon Quick Sight. This dashboard queries Iceberg metadata tables by using Spark SQL, and captures detailed metrics such as the number of active files and total storage. This information is then stored in AWS Glue tables for operational purposes. Finally, a Quick Sight dashboard, as shown in the following illustration, is created by using Amazon Athena. This information helps you identify and address specific problems in your systems.



The example Quick Sight dashboard collects the following key performance indicators (KPIs) for an Iceberg table:

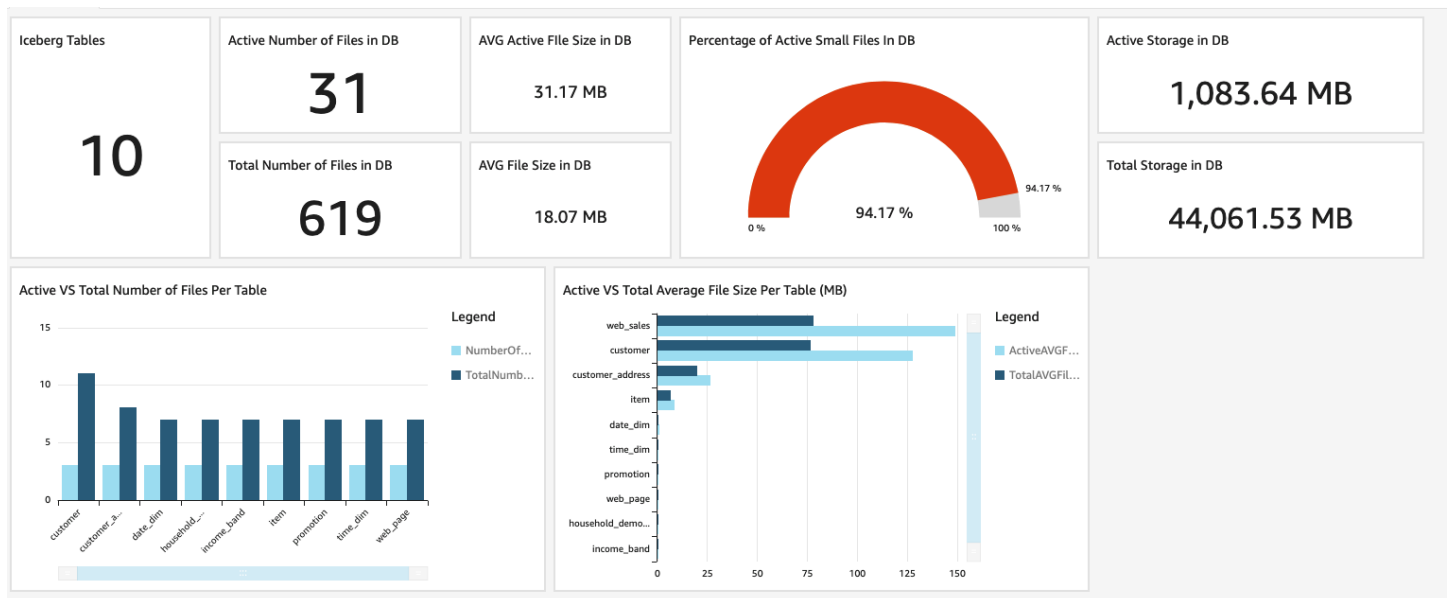
KPI	Description	Query
Number of files	The number of files in the Iceberg table (for all snapshots)	<pre>select count(*) from <catalog.database. table_name>.all_files</pre>
Number of active files	The number of active files in the last snapshot of the Iceberg table	<pre>select count(*) from <catalog.database. table_name>.files</pre>
Average file size	The average file size, in megabytes, for all files in the Iceberg table	<pre>select avg(file_ size_in_bytes)/100 0000 from <catalog.database. table_name>.all_files</pre>
Average active file size	The average file size, in megabytes, for the active files in the Iceberg table	<pre>select avg(file_ size_in_bytes)/100 0000 from <catalog.database. table_name>.files</pre>
Percentage of small files	The percentage of active files that are smaller than 100 MB	<pre>select cast(sum(case when file_size _in_bytes < 100000000 then 1 else 0 end)*100/ count(*) as decimal(1 0,2)) from <catalog.database. table_name>.files</pre>
Total storage size	The total size of all the files in the table, excluding orphaned files and Amazon S3 object versions (if enabled)	<pre>select sum(file_ size_in_bytes)/100 0000</pre>

KPI	Description	Query
Total active storage size	The total size of all files in the current snapshots of a given table	<pre>from <catalog.database.table_name>.all_files select sum(file_size_in_bytes)/1000000 from <catalog.database.table_name>.files</pre>

For more information about creating dashboards, see the [Quick Sight documentation](#).

Database-level monitoring

The following example shows a monitoring dashboard that was created in Quick Sight to provide an overview of database-level KPIs for a collection of Iceberg tables.



This dashboard collects the following KPIs:

KPI	Description	Query
Number of files	The number of files in the Iceberg database (for all snapshots)	This dashboard uses the table-level queries provided in the previous section and consolidates the outcomes.
Number of active files	The number of active files in the Iceberg database (based on the last snapshots of Iceberg tables)	
Average file size	The average file size, in megabytes, for all files in the Iceberg database	
Average active file size	The average file size, in megabytes, for all active files in the Iceberg database	
Percentage of small files	The percentage of active files that are smaller than 100 MB in the Iceberg database	
Total Storage size	The total size of all files in the database, excluding orphaned files and Amazon S3 object versions (if enabled)	
Total active storage size	The total size of all files in the current snapshots of all tables in the database	

Preventive maintenance

By setting up the monitoring capabilities discussed in the previous sections, you can approach table maintenance from a preventive instead of reactive angle. For example, you can use the table-level and database-level metrics to schedule actions such as the following:

- Use bin packing compaction to group small files when a table reaches N small files.
- Use bin packing compaction to merge delete files when a table reaches N delete files in a given partition.
- Remove small files that were already compacted by removing snapshots when the total storage is X times higher than active storage.

Governance and access control for Apache Iceberg on AWS

Apache Iceberg integrates with AWS Lake Formation to simplify data governance. This integration allows data lake administrators to assign cell-level access permissions to Iceberg tables. For an example of querying Iceberg tables by using Amazon Athena and AWS Lake Formation, see the AWS blog post [Interact with Apache Iceberg tables using Amazon Athena and cross account fine-grained permissions using AWS Lake Formation](#).

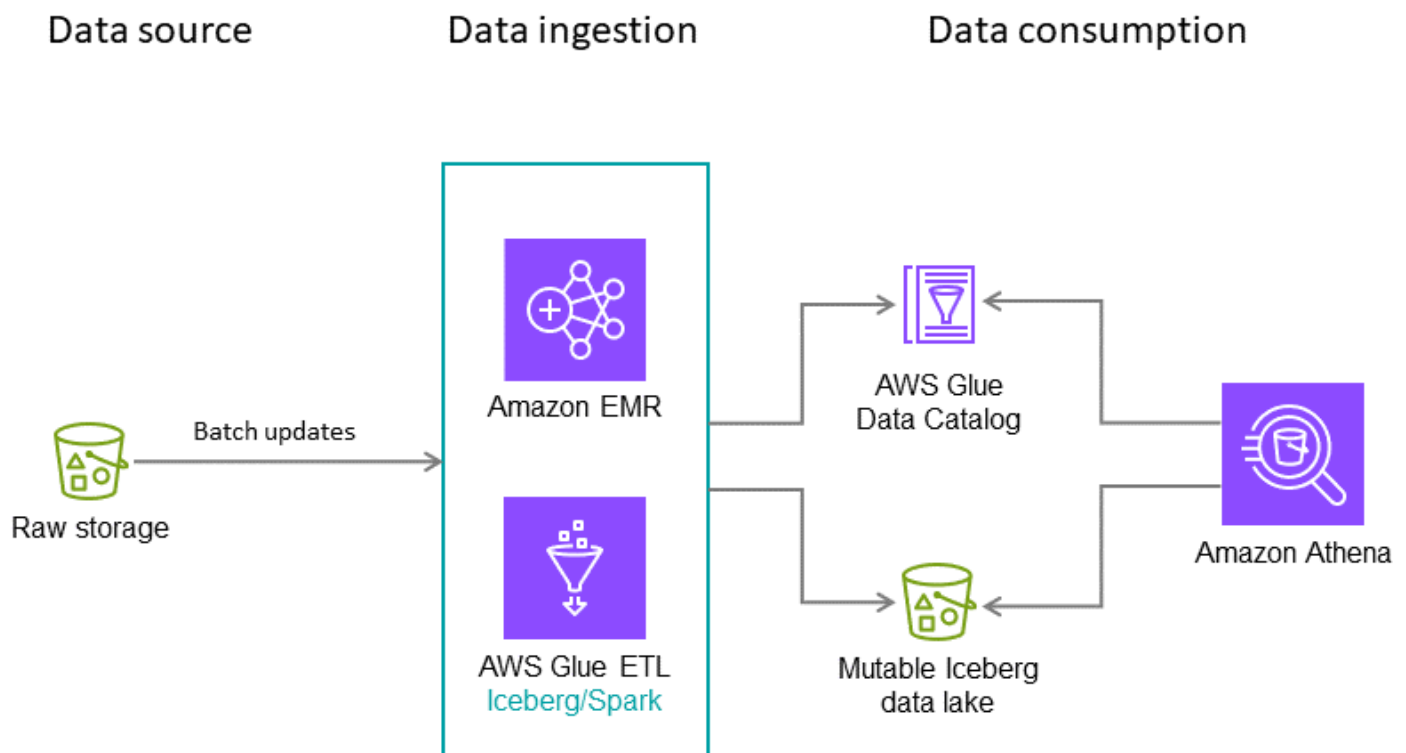
Reference architectures for Apache Iceberg on AWS

This section provides examples of how to apply best practices in different use cases such as batch ingestion and a data lake that combines batch and streaming data ingestion.

Nightly batch ingestion

For this hypothetical use case, let's say that your Iceberg table ingests credit card transactions on a nightly basis. Each batch contains only incremental updates, which must be merged into the target table. Several times per year, full historical data is received. For this scenario, we recommend the following architecture and configurations.

Note: This is just an example. The optimal configuration depends on your data and requirements.



Recommendations:

- File size: 128 MB, because Apache Spark tasks process data in 128 MB chunks.
- Write type: copy-on-write. As detailed earlier in this guide, this approach helps ensure that data is written in a read-optimized fashion.

- Partition variables: year/month/day. In our hypothetical use case, we query recent data most frequently, although we occasionally run full table scans for the past two years of data. The goal of partitioning is to drive fast read operations based on the requirements of the use case.
- Sort order: timestamp
- Data catalog: AWS Glue Data Catalog

Data lake that combines batch and near real-time ingestion

You can provision a data lake on Amazon S3 that shares batch and streaming data across accounts and Regions. For an architecture diagram and details, see the AWS blog post [Build a transactional data lake using Apache Iceberg, AWS Glue, and cross-account data shares using AWS Lake Formation and Amazon Athena](#).

Resources

- [Using the Iceberg framework in AWS Glue](#) (AWS Glue documentation)
- [Iceberg](#) (Amazon EMR documentation)
- [Using Apache Iceberg tables](#) (Amazon Athena documentation)
- [Amazon S3 documentation](#)
- [Quick documentation](#)
- [Glue Catalog and Lake Formation Permissions replication](#) (GitHub repository)
- [Apache Iceberg documentation](#)
- [Apache Spark documentation](#)

Contributors

The following people at AWS authored, co-authored, and reviewed this guide.

Contributors

- Stefano Sandona, Solutions Architect, Big Data
- Imtiaz (Taz) Sayed, Solutions Architect Tech Leader, Analytics
- Shana Schippers, Solutions Architect, Big Data
- Prashant Singh, Software Development Engineer, Amazon EMR
- Arun A K, Solutions Architect, Big Data and ETL
- Francisco Morillo, Solutions Architect, Streaming
- Suthan Phillips, Analytics Architect, Amazon EMR
- Sercan Karaoglu, Solutions Architect
- Yonatan Dolan, Analytics Specialist
- Guy Bachar, Solutions Architect
- Sofia Zilberman, Solutions Architect, Streaming
- Dan Stair, Specialist Solutions Architect
- Sakti Mishra, Solutions Architect
- Ron Ortloff, Principal Product Manager, Amazon S3
- David Zhang, Solutions Architect, Analytics

Reviewers

- Rick Sears, General Manager, Amazon EMR
- Linda OConnor, Amazon EMR Specialist
- Ian Meyers, Director, Amazon EMR
- Vinita Ananth, Director, Amazon EMR Product Management
- Jason Berkowitz, Product Manager, AWS Lake Formation
- Mahesh Mishra, Product Manager, Amazon Redshift
- Vladimir Zlatkin, Manager, Solutions Architecture, Big Data
- Karthik Prabhakar, Analytics Architect, Amazon EMR

- Vijay Jain, Product Manager
- Anupriti Warade, Product Manager, Amazon S3
- Ajit Tandale, Solutions Architect, Data
- Gwen Chen, Product Marketing Manager
- Noritaka Sekiyama, Principal Architect, Big Data

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an [RSS feed](#).

Change	Description	Date
New section	Added information about working with Iceberg table format specification version 3 .	November 26, 2025
Correction	Corrected information about <code>gc.enabled</code> settings in the snapshot procedure section.	October 24, 2025
Additions	Added new sections on working with Iceberg tables by using Trino and Pylceberg , and expanded the section on migrating tables to Iceberg .	August 12, 2025
Updates	Enhanced and clarified information throughout the guide to reflect the latest versions of AWS Glue, Amazon EMR, and Apache Iceberg.	July 14, 2025
Additions	Added a new section on working with Iceberg tables by using Amazon Data Firehose.	February 20, 2025
Initial publication	—	April 30, 2024