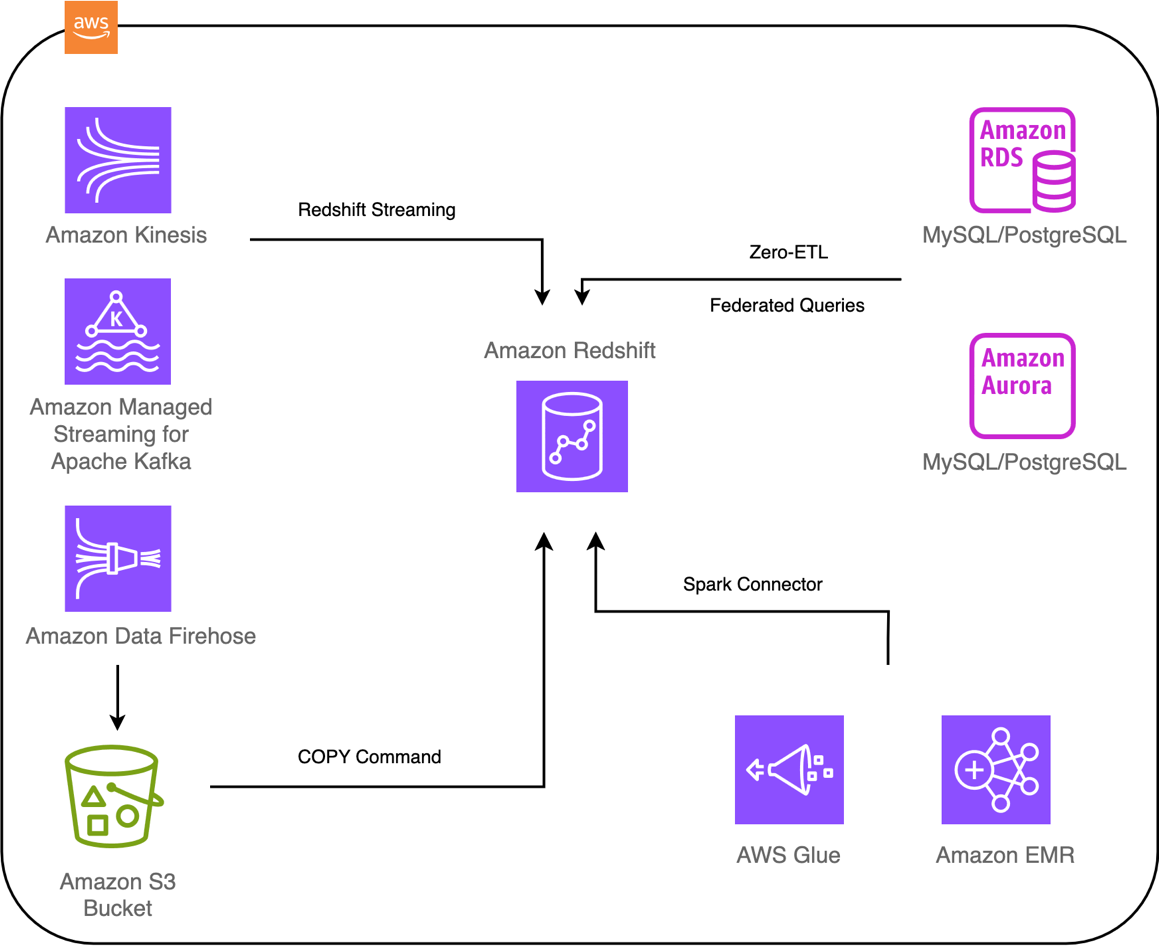
**Redshift Data Ingestion Options. What, Why, How?**

**Introduction**  
  
[Amazon Redshift](https://aws.amazon.com/redshift/), a warehousing service, offers a variety of options for ingesting data from diverse sources into its high-performance, scalable environment. Whether your data resides in operational databases, data lakes, on-premises systems, [Amazon Elastic Compute Cloud (Amazon EC2),](https://aws.amazon.com/ec2/) or other AWS services, Amazon Redshift provides multiple ingestion methods to meet your specific needs. The currently available choices include:

Amazon Redshift's [COPY command](https://docs.aws.amazon.com/redshift/latest/dg/r_COPY.html) for loading data from [Amazon Simple Storage Service](https://aws.amazon.com/s3/) (Amazon S3), [Amazon EMR](https://aws.amazon.com/emr/), [Amazon DynamoDB](https://aws.amazon.com/dynamodb/) or remote hosts over Secure Shell (SSH). This native feature of Redshift utilizes Redshift's massive parallel processing (MPP) to load objects directly from data sources into Redshift tables. Amazon Redshift [zero-ETL integrations](https://docs.aws.amazon.com/redshift/latest/mgmt/zero-etl-using.html) loads data from [Amazon Aurora](https://aws.amazon.com/rds/aurora/) MySQL, [Amazon Relational Database Service (Amazon RDS)](https://aws.amazon.com/rds/) MySQL, Amazon RDS PostgreSQL and DynamoDB with the added ability to perform transformations after loading. [Redshift federated queries](https://docs.aws.amazon.com/redshift/latest/dg/federated-overview.html) run queries using source database compute with the results returned to Redshift. The [Amazon Redshift integration for Apache Spark](https://docs.aws.amazon.com/redshift/latest/mgmt/spark-redshift-connector.html) combined with [AWS Glue](https://aws.amazon.com/glue/) or Amazon EMR performs transformations before loading data into Redshift. Finally, Redshift streaming supports ingestion of streaming sources including [Amazon Kinesis Data Streams](https://docs.aws.amazon.com/streams/latest/dev/introduction.html), [Amazon Managed Streaming for Apache Kafka](https://aws.amazon.com/msk/) and [Amazon Data Firehose](https://aws.amazon.com/firehose/).  
  
This article explores each option, determines which are suitable for different use-cases, and answers the questions of what, why, how when selecting a Redshift tool or feature for data ingestion.

Amazon Redshift data ingestion options

  
  
**Redshift COPY command**  
  
The [Redshift COPY command](https://aws.amazon.com/blogs/big-data/part-1-introducing-new-features-for-amazon-redshift-copy/), a simple low-code data ingestion tool, loads data into Redshift from Amazon S3, DynamoDB, EMR and remote hosts over Secure Shell (SSH). It is a fast and efficient way to load large datasets into Redshift. Leveraging Redshift's massively parallel processing (MPP) architecture to read and load large amounts of data in parallel from files or data from supported data sources. This allows users to utilize parallel processing by splitting data into multiple files, especially when the files are compressed. Recommended use cases for the COPY command include loading large data-sets and data from supported data sources. COPY automatically splits large uncompressed delimited text files into smaller scan ranges to utilize the parallelism of the Redshift provisioned clusters and serverless workgroups. With [Auto COPY](https://aws.amazon.com/about-aws/whats-new/2022/11/amazon-redshift-supports-auto-copy-amazon-s3/), automation enhances the COPY command by adding jobs for automatic ingestion of data.  
  
**Amazon Redshift federated queries**  
  
Amazon Redshift federated queries allows users to incorporate live data from Amazon RDS or Amazon Aurora operational databases as part of business intelligence and reporting applications. Federated queries are useful for use-cases where organizations want to combine data from their operational systems with data stored in Amazon Redshift. Federated queries allow query of data across Amazon RDS MySQL and PostgreSQL data sources without the need for Extract, Transform & Load (ETL) pipelines. If storing operational data in a data warehouse is a requirement, synchronization of tables between operational data stores and Amazon Redshift tables is supported. In scenarios where data transformation is required, leverage Redshift stored procedures to modify data in Amazon Redshift tables.

**Zero-ETL Integrations**  
  
Amazon Aurora zero-ETL to Amazon Redshift is a feature that allows access to operational data from Amazon Aurora MySQL (Amazon Aurora PostgreSQL and RDS MySQL in preview) from Amazon Redshift without the need for ETL in near-real time. Utilize Zero-ETL to simplify ingestion pipelines for performing CDC (change data capture) from an Amazon Aurora database to Amazon Redshift. Built on integration of Amazon Redshift and Amazon Aurora’s storage layers, zero-ETL boasts simple setup, data filtering, automated observability, auto-recovery and integration with either Amazon Redshift provisioned clusters or [Redshift serverless](https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-serverless.html) workgroups.  
  
**Amazon Redshift integration for Apache Spark**  
  
The Amazon Redshift integration for Apache Spark, automatically included through Amazon EMR or AWS Glue provides performance and security optimizations when compared to the community-provided connector. The integration enhances and simplifies security with [Amazon IAM](https://aws.amazon.com/iam/) authentication support. AWS Glue 4.0 provides a visual ETL tool for authoring jobs to read from and write to Amazon Redshift, leveraging the Redshift Spark connector for connectivity. This simplifies the process of building ETL pipelines to Amazon Redshift. The Spark Redshift connector allows use of Spark applications to process and transform data before loading into Amazon Redshift. The integration minimizes the manual process of setting up a Spark Redshift connector and shortens the time needed to prepare for analytics and machine learning (ML) tasks. It allows users to specify the connection to a data warehouse and start working with Amazon Redshift data from their Apache Spark-based applications within minutes.   
  
The integration provides push-down capabilities for sort, aggregate, limit, join, and scalar function operations to optimize performance by moving only the relevant data from Amazon Redshift to the consuming Apache Spark application. Spark jobs are suitable for data processing pipelines and the need to leverage Spark's advanced data transformation capabilities.   
  
**Amazon Redshift streaming ingestion**  
  
The key benefit of Amazon Redshift streaming ingestion is the ability to ingest hundreds of megabytes of data per second directly from streaming sources into Amazon Redshift with very low latency, supporting real-time analytics and insights. Supporting streams from Amazon Kinesis Data Streams, Amazon Managed Streaming for Apache Kafka, and Amazon Data Firehose, streaming ingestion requires no data staging, supports flexible schemas and is configured with SQL. Streaming ingestion powers real-time dashboards and operational analytics by directly ingesting data into Amazon Redshift materialized views.   
  
**Evaluating Amazon Redshift ingestion techniques**  
  
When building a robust and efficient data warehouse with Amazon Redshift, choosing the right data ingestion strategy is crucial. Amazon Redshift offers a variety of ingestion options, each tailored to different use cases and requirements. Understanding the benefits of these options helps optimize data workflows, meet business requirements, and reduce costs. This section explores the key benefits of six prominent Amazon Redshift ingestion methods: COPY command, Auto COPY, federated queries, streaming ingestion, zero-ETL integrations, integration for Apache Spark. Evaluating the strengths of each solution supports informed decisions that align with data integration needs and business objectives.  
  
**COPY Command (Data source: Amazon S3, use-case: large-scale batch data ingestion)**

* **Performance**: Efficiently load large datasets from Amazon S3 or other sources in parallel with optimized throughput.
* **Simplicity**: Straightforward and easy to use, requiring minimal setup.
* **Cost-optimized**: Utilize Amazon Redshift's massively parallel processing at a lower cost by reducing data transfer time.
* **Flexibility**: Supports file formats such as CSV, JSON, Parquet, ORC, and AVRO.

**Auto Copy (Data source: Amazon S3, use-case: Automated large-scale batch data ingestion)**

* **Automation**: Automatically ingest new data files as they arrive in Amazon S3, reducing manual intervention.
* **Event-Driven**: Utilize AWS Lambda and Amazon S3 event notifications to trigger data loads, ensuring near real-time data availability.
* **Scalability**: Manage large-scale data ingestion seamlessly without manual monitoring.
* **Efficiency**: Reduce latency in data availability by automating the ingestion process.

**Federated Query (Data source: Amazon RDS, Amazon Aurora, use-case: Query & JOIN data from operational databases)**

* **Real-Time access**: activate querying of live data across discrete sources, such as Amazon RDS and Amazon Aurora, with the need to move the data.
* **Unified data view**: Provides a single view of data across multiple databases, simplifying data analysis and reporting.
* **Cost savings**: Eliminate the need for ETL processes to move data into Redshift, saving on storage and compute costs.
* **Flexibility**: Supports Amazon RDS and Amazon Aurora data sources, offering flexibility in accessing and analyzing distributed data.

**Zero-ETL Integrations (Data source: Amazon Aurora, Amazon RDS, use-case: ETL pipeline simplification with data filtering from transactional databases)**

* + **Seamless integration**: Automatically integrates and synchronizes data between operational databases and Amazon Redshift without the need for custom ETL processes.
  + **Near real-time insights**: Provide near real-time data updates, ensuring the most current data is available for analysis.
  + **Ease of use**: Simplify data architecture by eliminating the need for separate ETL tools and processes.
  + **Efficiency**: Minimize data latency and ensure data consistency across systems, enhancing overall data accuracy and reliability.

**Integration for Apache Spark (Data source: Amazon S3, Amazon EMR, AWS Glue, use-case: Simplify building of ETL pipelines with data transformation requirements)**

* **High Performance**: Leverage the distributed computing power of Apache Spark for large-scale data processing and analysis.
* **Scalability**: Easily scale to handle massive datasets by distributing computation across multiple nodes.
* **Flexibility**: Support a wide range of data sources and formats, providing versatility in data processing tasks.
* **Interoperability**: Seamlessly integrate with Amazon Redshift for efficient data transfer and query execution.

**Streaming Ingestion (Data source: Amazon Kinesis Data Streams, Amazon Managed Streaming for Apache Kafka, Amazon Data Firehose, use-case: near real-time streaming analytics)**

* **Low Latency**: Ingest streaming data in near real-time, making streaming ingestion ideal for time-sensitive applications such as IoT, financial transactions, and clickstream analysis.
* **Scalability**: Manage high throughput and large volumes of streaming data from sources such as **Amazon Kinesis Data Streams, Amazon Managed Streaming for Apache Kafka, and Amazon Data Firehose.**
* **Integration**: Easily integrate with other AWS services to build end-to-end streaming data pipelines.
* **Continuous Updates**: Keep data in Redshift continuously updated with the latest information from the data streams.

**Amazon Redshift ingestion use-cases and examples**

**Redshift COPY command use-case: application log data ingestion and analysis**

Ingesting application log data stored in Amazon S3 is a common use-case for Redshift’s COPY command. Data engineers in an organization need to analyze application log data to gain insights into user behavior, identify potential issues, and optimize a platform’s performance. To achieve this, use the COPY command in Amazon Redshift to ingest log data from Amazon S3 into Amazon Redshift tables.

By using the COPY command, data engineers load log data in parallel from multiple files stored in Amazon S3 buckets into Amazon Redshift tables. This parallelization makes use of Amazon Redshift's massively parallel processing (MPP) architecture, allowing for faster data ingestion compared to other ingestion methods.

A simple example of the COPY command loading data from a set of CSV files in an S3 bucket into a Redshift table:

COPY myschema.mytable

FROM 's3://my-bucket/data/files/'

IAM\_ROLE ‘arn:aws:iam::1234567891011:role/MyRedshiftRole’

FORMAT AS CSV;

In this example:

* myschema.mytable is the target Redshift table for data load.
* 's3://my-bucket/data/files/' is the S3 path where the CSV files are located.
* IAM\_ROLE specifies the Amazon IAM role required to access the Amazon S3 bucket.
* FORMAT AS CSV specifies that the data files are in CSV format.

In addition to Amazon S3, The COPY command loads data from other sources such as Amazon DynamoDB, Amazon EMR, remote hosts via SSH, or other Redshift databases. COPY provides options to specify data formats, delimiters, compression, and other parameters to handle different data sources and formats. To get started with Amazon Redshift’s COPY command, see [Using the COPY command to load from Amazon S3](https://docs.aws.amazon.com/redshift/latest/dg/t_loading-tables-from-s3.html).

**Amazon Redshift integration for Apache Spark use-case: gaming player events written to Amazon S3**

Consider a large volume of gaming player events stored in Amazon S3. The events require data transformation, cleansing and preprocessing to extract insights, generate reports, or build machine learning models. In this case, utilize the scalability and processing power of Amazon EMR to perform required data changes using Apache Spark. Once processed, the transformed data must be loaded into Amazon Redshift for further analysis, reporting and integration with business intelligence (BI) tools. In this scenario, leverage Amazon Redshift integration for Apache Spark to perform the necessary data transformations and load the processed data into Amazon Redshift.

The following implementation example assumes gaming player events in parquet format stored in Amazon S3 (s3://<bucket\_name>/player\_events/)

1. Set up the environment:

a. Launch an Amazon EMR (emr-6.9.0) cluster with Apache Spark (Spark 3.3.0) with Amazon Redshift integration with Apache Spark support.

b. Configure the necessary Amazon IAM role for accessing Amazon S3 and Amazon Redshift.

1. Create a Spark job that sets up a connection to Amazon Redshift, reads data from Amazon S3, performs transformations and writes resulting data to Amazon Redshift.

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, lit

# Create a SparkSession

spark = SparkSession.builder \

.appName("RedshiftSparkJob") \

.getOrCreate()

# Set Amazon Redshift connection properties

redshift\_jdbc\_url = "jdbc:redshift://<redshift-endpoint>:<port>/<database>"

redshift\_table = "<schema>.<table\_name>"

temp\_s3\_bucket = "s3://<bucket\_name>/temp/"

iam\_role\_arn = "<iam\_role\_arn>"

# Read data from Amazon S3

s3\_data = spark.read.format("parquet") \

.load("s3://<bucket\_name>/player\_events/")

# Perform transformations

transformed\_data = s3\_data.withColumn("transformed\_column", lit("transformed\_value"))

# Write the transformed data to Amazon Redshift

transformed\_data.write \

.format("io.github.spark\_redshift\_community.spark.redshift") \

.option("url", redshift\_jdbc\_url) \

.option("dbtable", redshift\_table) \

.option("tempdir", temp\_s3\_bucket) \

.option("aws\_iam\_role", iam\_role\_arn) \

.mode("overwrite") \

.save()

In this example, first import the necessary modules and create a SparkSession. Set the connection properties for Amazon Redshift, including the endpoint, port, database, schema, table name, temporary S3 bucket path, and the IAM role ARN for authentication. Read data from Amazon S3 in parquet format using the  spark.read.format("parquet").load()  method. Perform a transformation on the Amazon S3 data by adding a new column  transformed\_column with a constant value using the withColumn  method and the lit function. Write the transformed data to Amazon Redshift using the  write  method and the io.github.spark\_redshift\_community.spark.redshift format. Set the necessary options for the Redshift connection URL, table name, temporary S3 bucket path, and IAM role ARN.  Use the mode("overwrite") option to overwrite the existing data in the Amazon Redshift table with the transformed data. To get started with Amazon Redshift integration for Apache Spark, see [Amazon Redshift integration for Apache Spark.](https://docs.aws.amazon.com/redshift/latest/mgmt/spark-redshift-connector.html) For more examples of using the Amazon Redshift for Apache Spark Connector, see [Amazon Redshift Integration with Apache Spark](https://aws.amazon.com/blogs/aws/new-amazon-redshift-integration-with-apache-spark/).

**Amazon Redshift streaming ingestion use-case: IoT telemetry near real-time analysis**

Imagine a fleet of IoT devices (sensors and industrial equipment) that generate a continuous stream of telemetry data such as temperature readings, pressure measurements, or operational metrics. Ingesting this data in real-time to perform analytics to monitor the devices, detect anomalies, and make data-driven decisions requires a streaming solution integrated with an Amazon Redshift data warehouse.

In this example, use Amazon Managed Streaming for Apache Kafka as the streaming source for IoT telemetry data.

1. Create an external schema in Amazon Redshift
   1. Connect to an Amazon Redshift cluster using a SQL client or Query Editor v2 in the Amazon Redshift console.
   2. Create an external schema that references the Amazon MSK cluster.

CREATE EXTERNAL SCHEMA kafka\_schema

FROM KAFKA

BROKER 'broker-1.example.com:9092,broker-2.example.com:9092'

TOPIC 'iot-telemetry-topic'

REGION 'us-east-1'

IAM\_ROLE 'arn:aws:iam::123456789012:role/RedshiftRoleForMSK';

1. Create a materialized view in Amazon Redshift
   1. Define a materialized view that maps the Kafka topic data to Amazon Redshift table columns.
   2. CAST the streaming message payload data type to Amazon Redshift’s SUPER type.
   3. Set the materialized view to auto refresh.

CREATE MATERIALIZED VIEW iot\_telemetry\_view

AUTO REFRESH YES

AS SELECT

kafka\_partition,

kafka\_offset,

kafka\_timestamp\_type,

kafka\_timestamp,

CAST(kafka\_value AS SUPER) payload

FROM kafka\_schema.iot-telemetry-topic;

1. Query the materialized view
   1. Now query the iot\_telemetry\_view materialized view to access the real-time IoT telemetry data ingested from the Kafka topic.
   2. The materialized view will automatically refresh as new data arrives in the Kafka topic.

SELECT

kafka\_timestamp,

payload:device\_id,

payload:temperature,

payload:pressure

FROM iot\_telemetry\_view;

By following this implementation, near real-time analytics on IoT device telemetry data using Amazon Redshift streaming ingestion with Amazon MSK is achieved. As telemetry data is received by an Amazon MSK topic, Amazon Redshift automatically ingests and reflects the data in a materialized view, supporting query and analysis of the data in near real-time. To get started with Amazon Redshift streaming ingestion, see [Amazon Redshift Streaming Ingestion](https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-streaming-ingestion.html). To learn more about streaming and customer use-cases, see [Amazon Redshift Streaming Ingestion](https://aws.amazon.com/redshift/redshift-streaming-ingestion/).

**Amazon Redshift federated queries use case: integrated reporting and analytics for a retail company**

A retail company has an operational database running on Amazon RDS for PostgreSQL, which stores real-time sales transactions, inventory levels, and customer information data.  Additionally, a data warehouse runs on Amazon Redshift storing historical data for reporting and analytics purposes. To create an integrated reporting solution that combines real-time operational data with historical data in the data warehouse, without the need for multi-step ETL processes, follow these implementation steps.

1. Set up Network Connectivity
   1. Ensure that your Amazon Redshift cluster and Amazon RDS for PostgreSQL instance are in the same VPC or have network connectivity established through

[Amazon Virtual Private Cloud](https://aws.amazon.com/vpc/) (VPC) [peering](https://docs.aws.amazon.com/whitepapers/latest/aws-vpc-connectivity-options/vpc-peering.html), [AWS PrivateLink](https://aws.amazon.com/privatelink/) or [AWS Transit Gateway](https://aws.amazon.com/transit-gateway/).

1. Create a Secret and Amazon IAM Role for federated queries.
   1. In [AWS Secrets Manager](https://aws.amazon.com/secrets-manager/), create a new secret to store the credentials (username and password) for your Amazon RDS for PostgreSQL instance.
   2. Create an Amazon IAM role with permissions to access the AWS Secrets Manager secret and the Amazon RDS for PostgreSQL instance.
   3. Associate the Amazon IAM role with your Amazon Redshift cluster.
2. Create an External Schema in Redshift
   1. Connect to your Redshift cluster using a SQL client or Query Editor V2 in the Redshift console.
   2. Create an external schema that references your Amazon RDS for PostgreSQL instance:

CREATE EXTERNAL SCHEMA postgres\_schema

FROM POSTGRES

DATABASE 'mydatabase'

SCHEMA 'public'

URI 'endpoint-for-your-rds-instance.aws-region.rds.amazonaws.com:5432'

IAM\_ROLE 'arn:aws:iam::123456789012:role/RedshiftRoleForRDS'

SECRET\_ARN 'arn:aws:secretsmanager:aws-region:123456789012:secret:my-rds-secret-abc123';

1. Query the Operational Data Using Federated Queries
   1. Query tables in your Amazon RDS for PostgreSQL instance directly from Amazon Redshift using federated queries:

SELECT

r.order\_id,

r.order\_date,

r.customer\_name,

r.total\_amount,

h.product\_name,

h.category

FROM

postgres\_schema.orders r

JOIN redshift\_schema.product\_history h ON r.product\_id = h.product\_id

WHERE

r.order\_date >= '2024-01-01';

1. Create Views or Materialized Views for Reporting
   1. Create views or materialized views in Redshift that combine the operational data from federated queries with the historical data in Amazon Redshift for reporting purposes:

CREATE MATERIALIZED VIEW sales\_report AS

SELECT

r.order\_id,

r.order\_date,

r.customer\_name,

r.total\_amount,

h.product\_name,

h.category,

h.historical\_sales

FROM

(

SELECT

order\_id,

order\_date,

customer\_name,

total\_amount,

product\_id

FROM

postgres\_schema.orders

) r

JOIN redshift\_schema.product\_history h ON r.product\_id = h.product\_id;

By following this implementation, federated queries in Amazon Redshift integrate real-time operational data from Amazon RDS for PostgreSQL instances with historical data in an Amazon Redshift data warehouse. This approach eliminates the need for multi-step ETL processes and enables you to create comprehensive reports and analytics that combine data from multiple sources. To get started with Amazon Redshift federated queries ingestion, see [Querying data with federated queries in Amazon Redshift](https://docs.aws.amazon.com/redshift/latest/dg/federated-overview.html).

**Amazon Redshift zero-ETL integrations use-case: near real-time analytics for an e-commerce application**

Suppose an ecommerce application built on Amazon Aurora MySQL manages online orders, customer data, and product catalogs.

To perform near real-time analytics with data filtering on transactional data to gain insights into customer behavior, sales trends, and inventory

management without the overhead of building and maintaining multi-step ETL pipelines, use zero-ETL integrations for Amazon Redshift.

Implementation Steps:

1. Set up an Amazon Aurora MySQL Cluster
   1. Create an Amazon Aurora MySQL cluster in your desired AWS region.
   2. Configure the cluster settings, such as the instance type, storage, and backup options.
2. Create a zero-ETL integration with Amazon Redshift
   1. In the Amazon RDS console, navigate to the "zero-ETL integrations" section.
   2. Click "Create integration" and select your Aurora MySQL cluster as the source.
   3. Choose an existing or create a new Amazon Redshift cluster as the target.
   4. Provide a name for the integration and review the settings.
   5. Click "Create integration" to initiate the zero-ETL integration process.
3. Verify the integration status
   1. After the integration is created, monitor status in the RDS console or by querying the SVV\_INTEGRATION and

SYS\_INTEGRATION\_ACTIVITY system views in Amazon Redshift.

* 1. Wait for the integration to reach the "Active" state, indicating that data is being replicated from Amazon Aurora to Amazon Redshift.

1. Create analytics views
   1. Connect to your Amazon Redshift cluster using a SQL client or the [query editor v2](https://docs.aws.amazon.com/redshift/latest/mgmt/query-editor-v2-using.html)in the Amazon Redshift console.
   2. Create views or materialized views that combine and transform the replicated data from Amazon Aurora for your analytics use cases.

CREATE MATERIALIZED VIEW orders\_summary AS

SELECT

o.order\_id,

o.customer\_id,

SUM(oi.quantity \* oi.price) AS total\_revenue,

MAX(o.order\_date) AS latest\_order\_date

FROM

aurora\_schema.orders o

JOIN aurora\_schema.order\_items oi ON o.order\_id = oi.order\_id

GROUP BY

o.order\_id,

o.customer\_id;

1. Query and analyze the data
   1. Query the views or materialized views in Redshift to perform near real-time analytics on the transactional data from your Aurora MySQL cluster.

SELECT

customer\_id,

SUM(total\_revenue) AS total\_customer\_revenue,

MAX(latest\_order\_date) AS most\_recent\_order

FROM

orders\_summary

GROUP BY

customer\_id

ORDER BY

total\_customer\_revenue DESC;

This implementation easily achieves near real-time analytics for an e-commerce application's transactional data using the zero-ETL integration between Amazon Aurora MySQL and Amazon Redshift. The data automatically replicates from Amazon Aurora to Amazon Redshift, eliminating the need for multi-step ETL pipelines and supporting insights from the latest data quickly. To get started with Amazon Redshift zero-ETL integrations, see [Working with zero-ETL integrations](https://docs.aws.amazon.com/redshift/latest/mgmt/zero-etl-using.html). To learn more about Amazon Aurora zero-ETL integrations with Amazon Redshift, see [Amazon Aurora zero-ETL integrations with Amazon Redshift](https://aws.amazon.com/rds/aurora/zero-etl/).

**Conclusion**  
  
In summary, the choice of data ingestion method depends on factors such as the size and structure of data, the need for real-time access or transformations, data-sources, existing infrastructure, ease of use, and user skill-sets. Zero-ETL integrations and federated queries are suitable for simple data ingestion tasks or joining data between operational databases and Amazon Redshift analytics data. Large-scale data ingestion with transformation and orchestration benefit from Amazon Redshift integration with Apache Spark with Amazon EMR and AWS Glue. Bulk loading of data into Amazon Redshift regardless of data-set size fits perfectly with the capabilities of the Amazon Redshift COPY command. Utilizing streaming sources such as Amazon Kinesis Data Streams, Amazon Managed Streaming for Apache Kafka, or Amazon Data Firehose are ideal scenarios for utilizing Amazon streaming services integration for data ingestion.

This article detailed the options available for Amazon Redshift data ingestion. Please evaluate the features and guidance provided for your data ingestion workloads and let us know your feedback in the comments.