Apache Iceberg: Modern Table Format for Big Data Analytics

Iceberg is an open table format that brings ACID transactions, time travel, and schema evolution to big data. It works seamlessly with Spark, Trino, Flink, and more.

Rr by Rr Jj

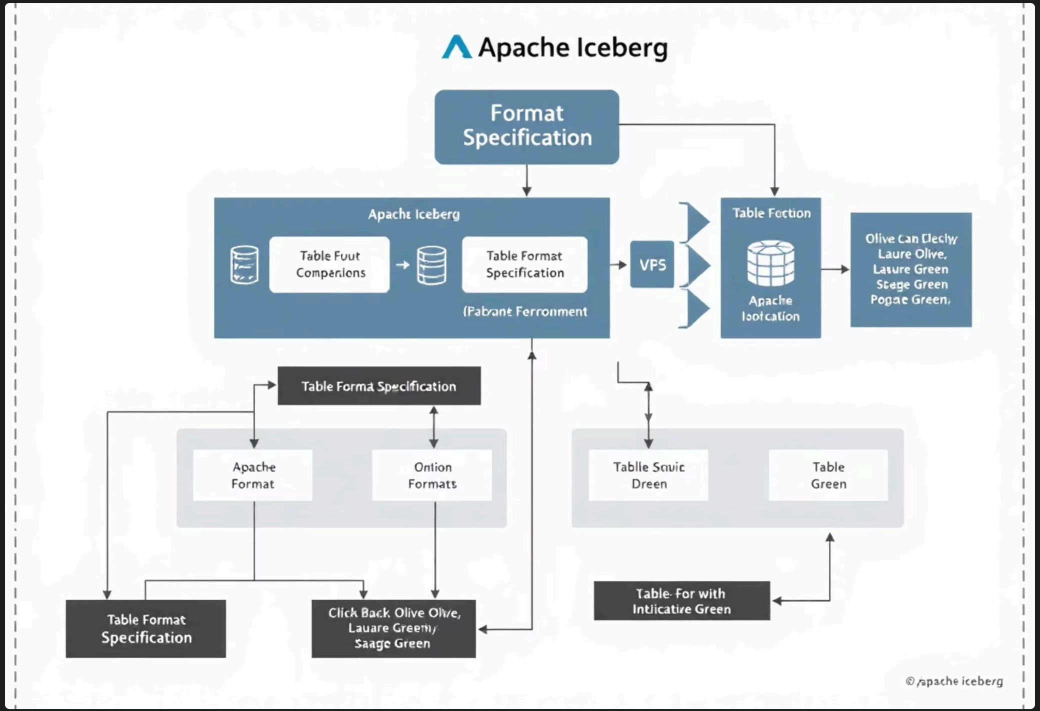
Understanding Iceberg: What It Is and Is Not

Iceberg IS

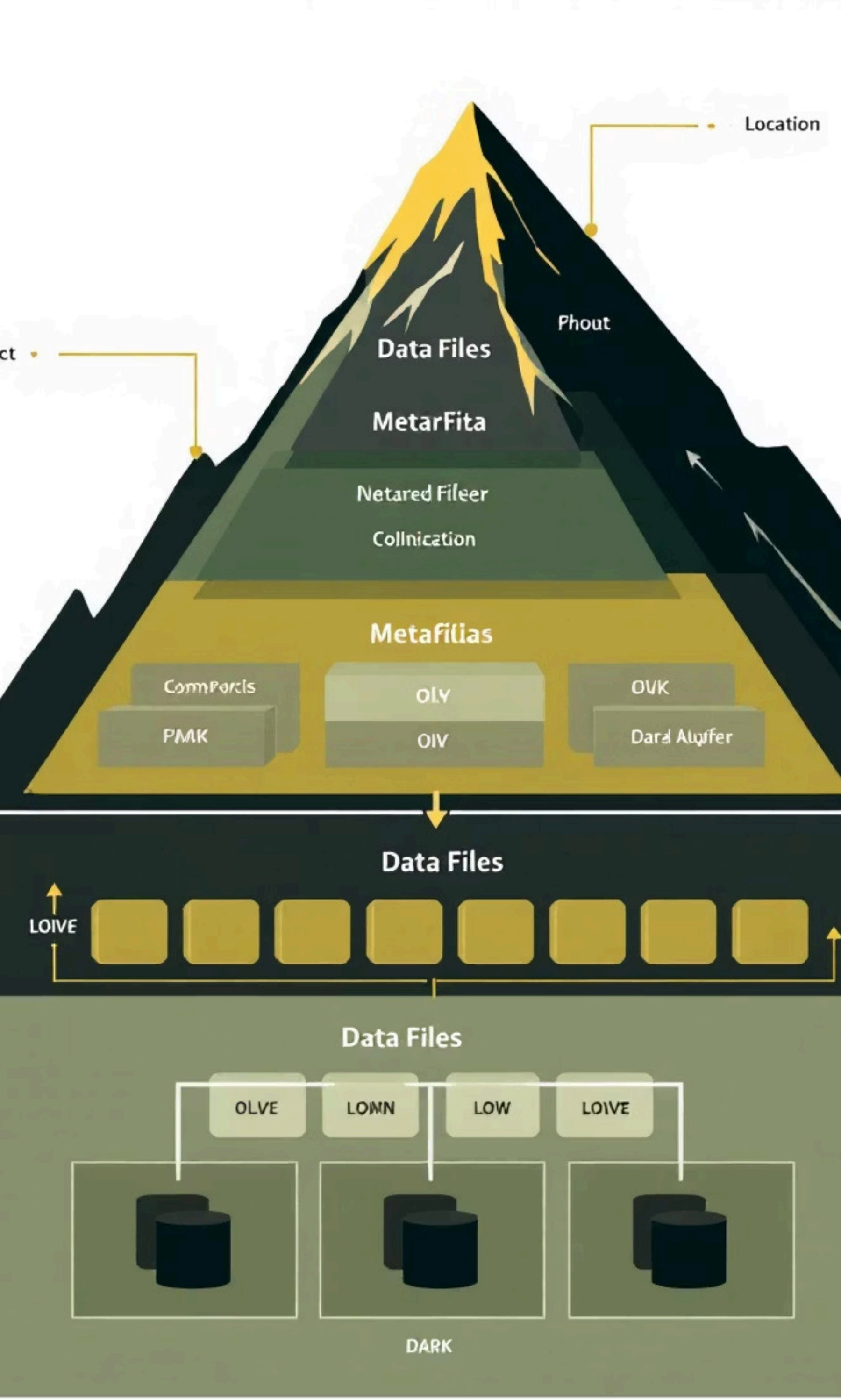
- A table format specification
- A collection of APIs and libraries
- A framework for data management in data lakes
- An interface for applications to interact with table data

Iceberg IS NOT

- A storage engine or file system
- An execution engine (like Spark or Flink)
- A database management system
- A standalone service or application



Iceberg provides a standardised way to manage table metadata, enabling advanced features while working with your existing processing engines and storage systems.



Understanding Apache Iceberg: Key Features and Benefits

Schema Evolution

Add, drop, rename or update columns without rebuilding tables. Changes are tracked and versioned for consistent reads.

Time Travel

Query data at specific points in time. Perfect for compliance, auditing, and reproducing previous analyses.

ACID Transactions

Full support for concurrent reads and writes with atomicity, consistency, isolation, and durability guarantees.

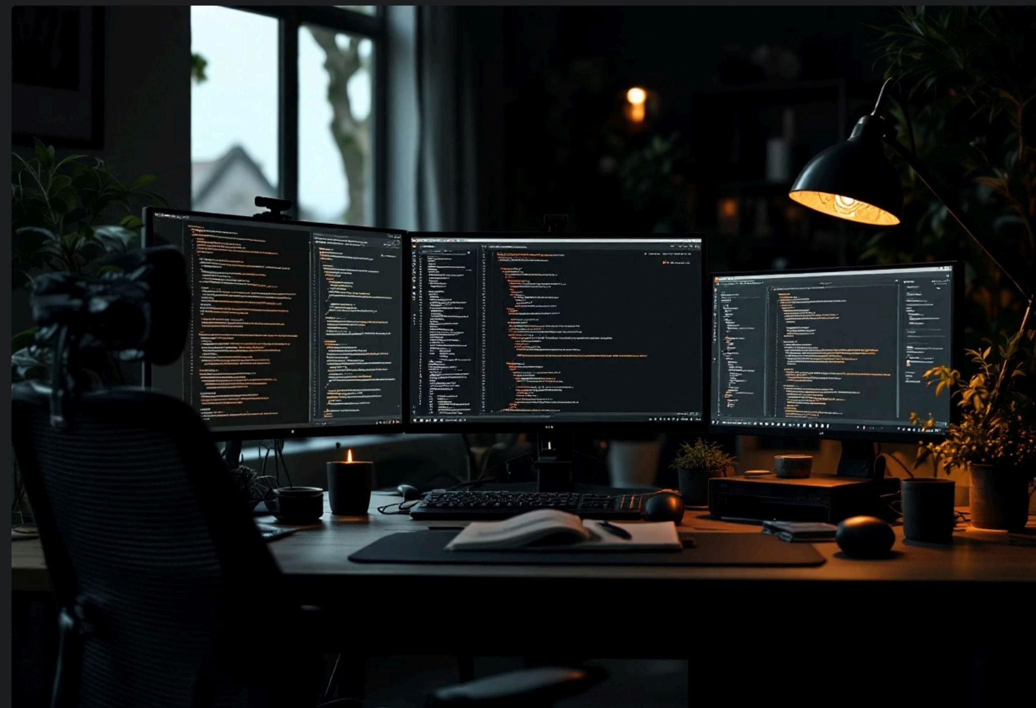
Getting Started: Setting Up Your Environment for Iceberg

Prerequisites

- Java 8 or higher
- Apache Spark 3.0+
- Python 3.6+ for PySpark
- S3, HDFS or other storage

Installation Options

- Maven dependencies
- Pre-built binaries
- Docker images



```
# Sample pip install
pip install pyspark
pip install pyiceberg
```

Configuring PySpark with Apache Iceberg

Add Iceberg Dependencies

Include Iceberg JARs in your Spark configuration using packages option.

```
pyspark --packages org.apache.iceberg:iceberg-spark3-runtime:1.3.0
```

Configure Spark Session

Set up Spark with Iceberg catalog and other necessary configurations.

```
spark = SparkSession.builder \
  .appName("Iceberg Example") \
  .config("spark.sql.extensions", "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
  .config("spark.sql.catalog.spark_catalog", "org.apache.iceberg.spark.SparkCatalog") \
  .config("spark.sql.catalog.spark_catalog.type", "hive") \
  .getOrCreate()
```

Verify Installation

Test your configuration by running a simple Iceberg query or command.

```
# Test that Iceberg is properly configured
spark.sql("SELECT 1").show()
spark.sql("CREATE DATABASE IF NOT EXISTS iceberg_db")
```

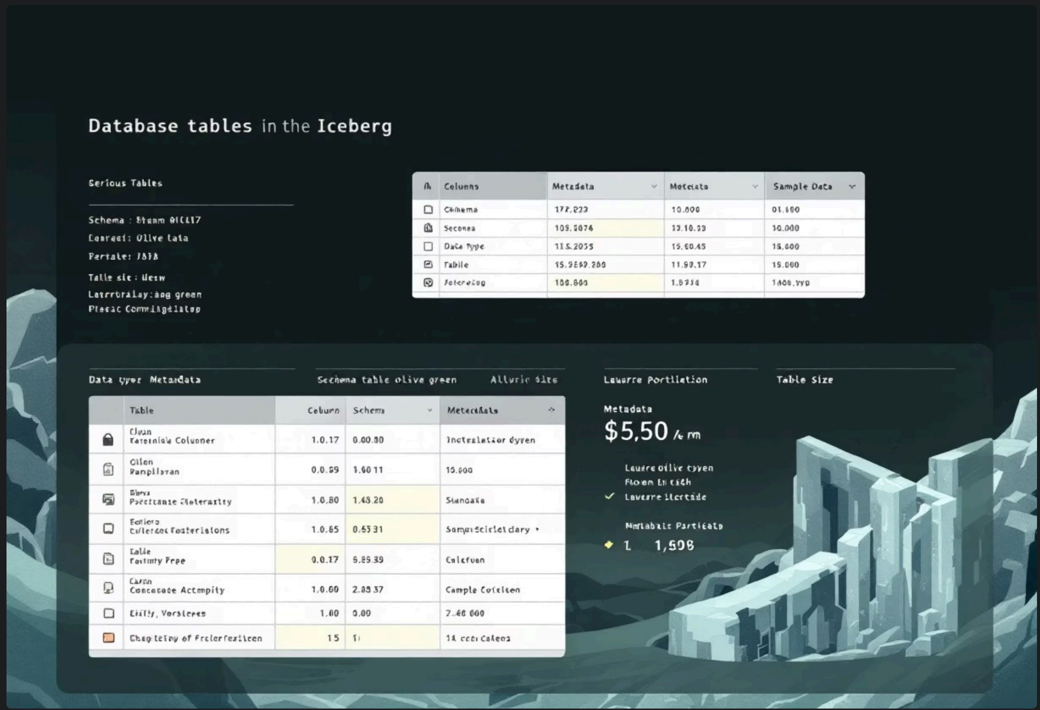
Creating and Managing Iceberg Tables with PySpark

Creating Tables

```
# SQL approach
spark.sql("""
CREATE TABLE iceberg_db.customers (
  id INT,
  name STRING,
  email STRING
) USING iceberg
""")

# DataFrame API approach
df = spark.createDataFrame(
  [(1, "John", "john@example.com")],
  ["id", "name", "email"]
)
df.writeTo("iceberg_db.customers").create()
```

Managing Tables

- List all tables in a namespace
- View table details and history
- Alter table properties
- Drop tables with clean metadata



Performing CRUD Operations on Iceberg Tables

Create/Insert

```
# Insert new data
spark.sql("""
INSERT INTO iceberg_db.customers
VALUES (2, 'Jane', 'jane@example.com')
""")

# DataFrame API
df.writeTo("iceberg_db.customers").append()
```

Read/Query

```
# SQL query
spark.sql("SELECT * FROM
iceberg_db.customers").show()

# DataFrame API
df =
spark.read.format("iceberg").load("iceberg_db.customers")
```

Update

```
# SQL update
spark.sql("""
UPDATE iceberg_db.customers
SET email = 'john.new@example.com'
WHERE id = 1
""")
```

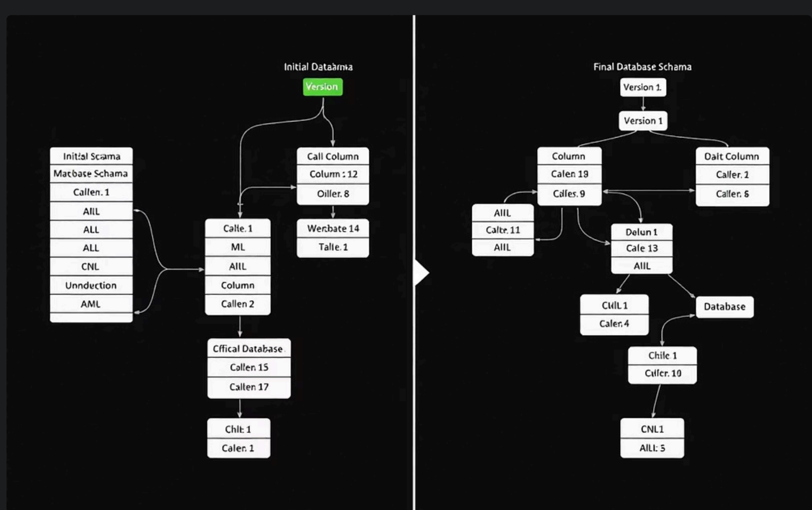
Delete

```
# SQL delete
spark.sql("""
DELETE FROM iceberg_db.customers
WHERE id = 2
""")
```

Advanced Features: Time Travel, Schema Evolution and Partitioning

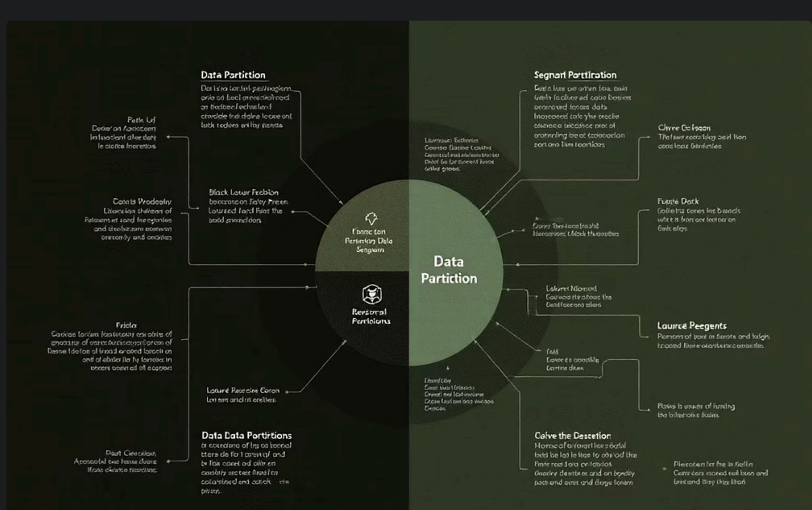
Time Travel

```
# Query as of timestamp
spark.read.option(
  "as-of-timestamp",
  "1662004800000"
).format("iceberg").load("iceberg_db.customers")

# Query by snapshot ID
spark.read.option(
  "snapshot-id",
  "8240436316246829840"
).format("iceberg").load("iceberg_db.customers")
```



Schema Evolution

```
# Add a new column
spark.sql("""
ALTER TABLE iceberg_db.customers
ADD COLUMN age INT
""")

# Rename a column
spark.sql("""
ALTER TABLE iceberg_db.customers
RENAME COLUMN email TO contact_email
""")
```



Partitioning

```
# Create partitioned table
spark.sql("""
CREATE TABLE iceberg_db.events (
  id INT,
  user_id INT,
  event_date DATE,
  event_type STRING
) USING iceberg
PARTITIONED BY (days(event_date), event_type)
""")
```

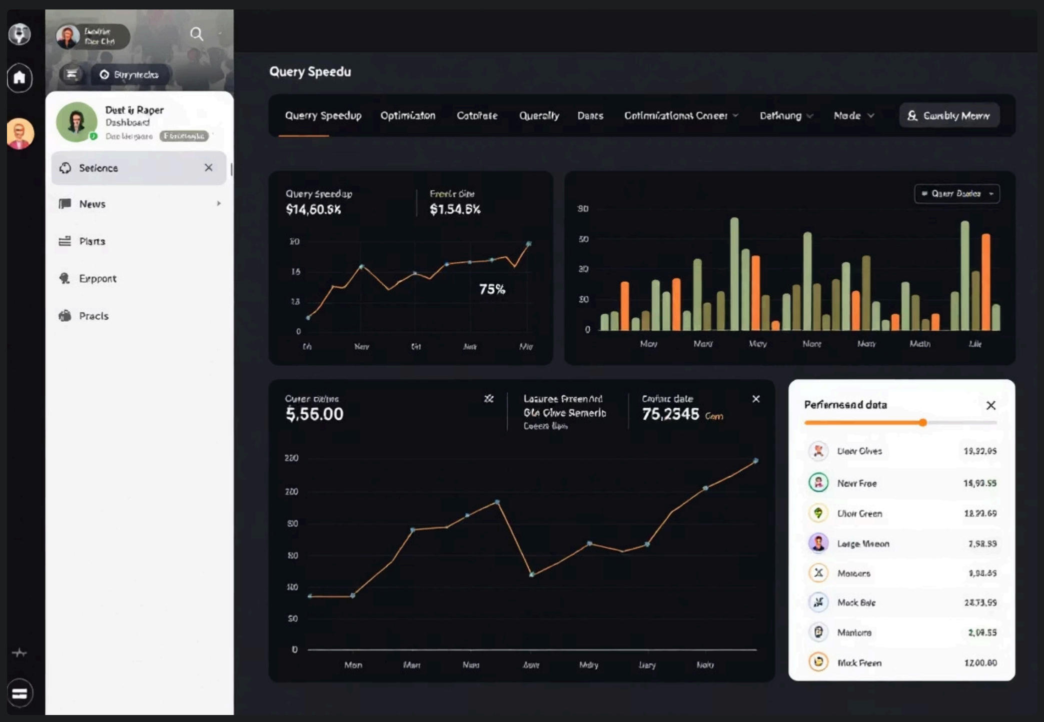
Performance Optimisation Techniques with Iceberg and PySpark

Data File Optimisations

- Compaction to merge small files
- Sorting for efficient filtering
- Strategic partitioning schemes

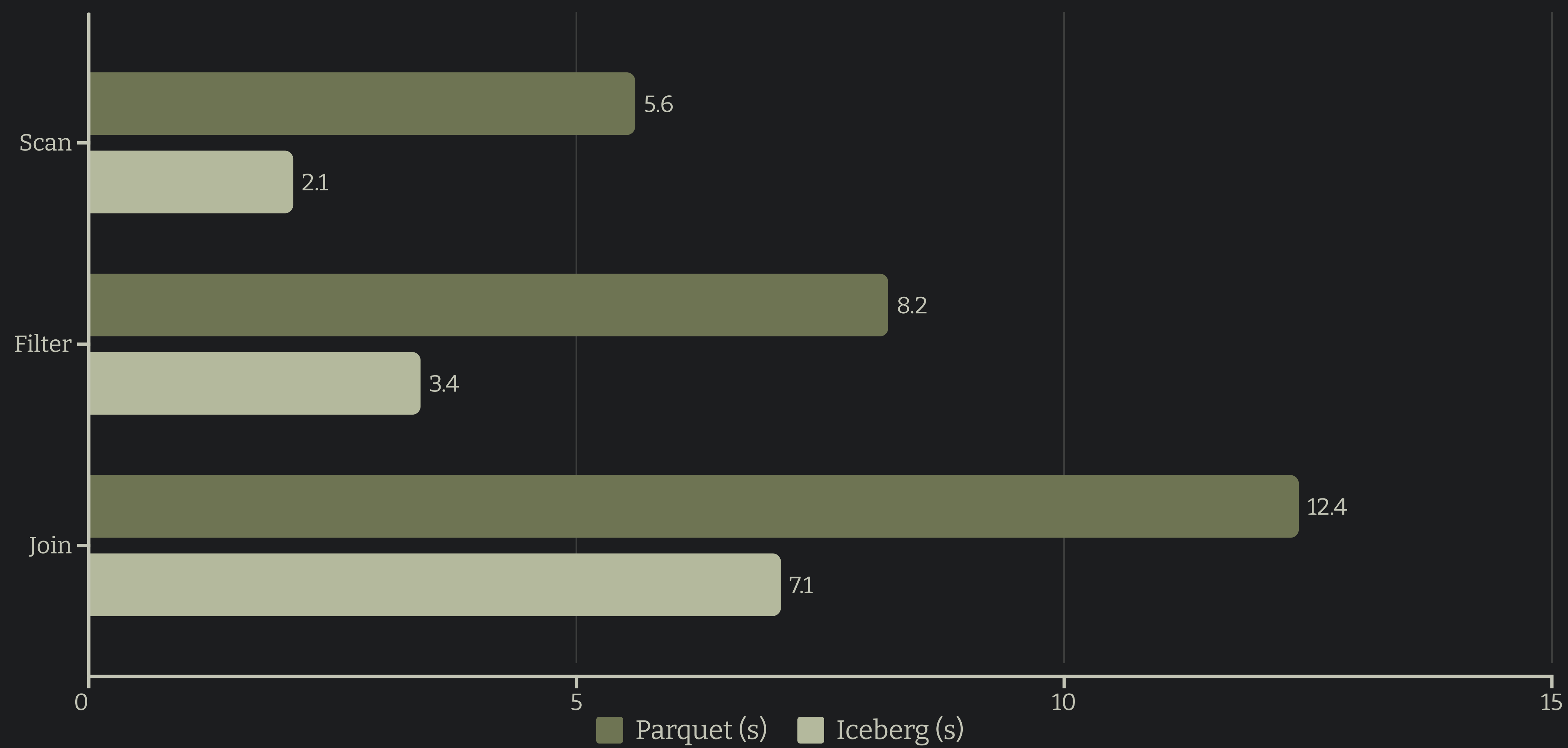
Query Optimisations

- Metadata filtering
- Partition pruning
- Column projection



```
# Compact small files
spark.sql("""
CALL spark_catalog.system.rewrite_data_files(
  table => 'iceberg_db.customers',
  options => map('min-input-files','5')
)
""")
```

Query Performance Comparison: Parquet vs Iceberg



Case Study: Real-world Implementation and Best Practices

E-commerce Data Platform

A major retailer migrated 5PB of data to Iceberg, reducing query times by 60% and enabling real-time analytics.

Their architecture connects PySpark jobs to Iceberg tables for both batch and streaming workloads.

Best Practices

- Keep metadata files small
- Use hidden partitioning
- Implement regular maintenance
- Monitor snapshot expiration

Common Pitfalls

- Over-partitioning tables
- Ignoring file size distribution
- Neglecting metadata cleanup
- Missing catalog backups

Ready to migrate? Start with a small dataset, measure performance, and gradually expand your Iceberg adoption.