Apache Iceberg: Modern Table Format for Big Data Analytics

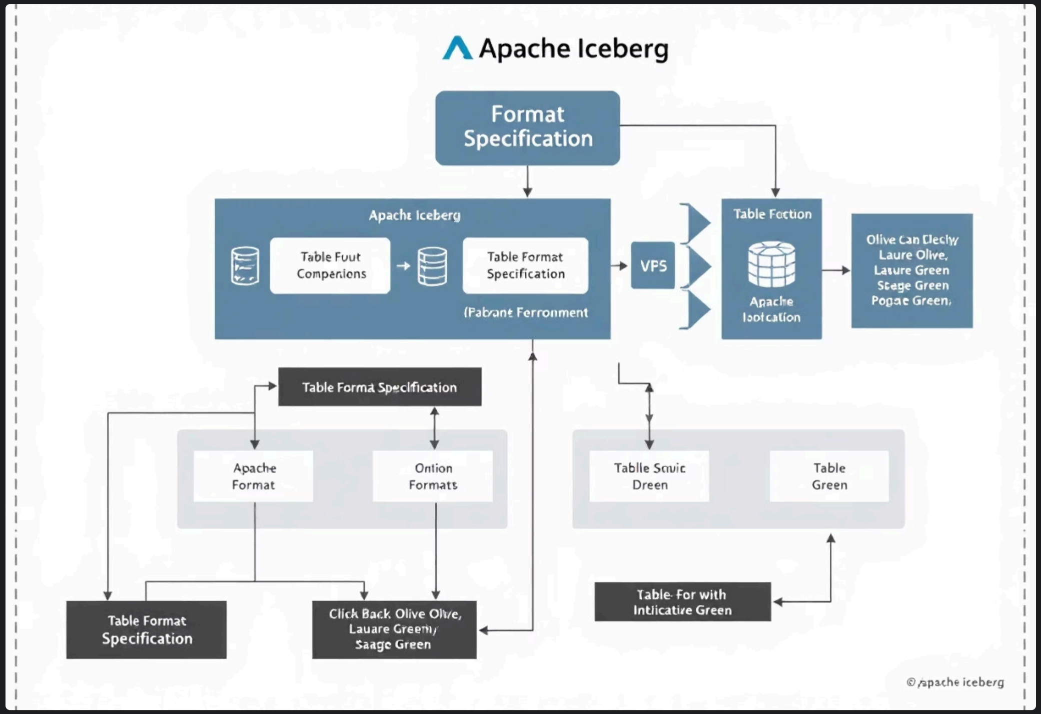
Iceberg is an open table format that brings ACID transactions, time travel, and schema evolution to big data. It works seamlessly with Spark, Trino, Flink, and more.

Redencio Jozefzoon - Sr. AI Engineer - Orange DSE

Understanding Iceberg: What It Is and Is Not

Iceberg IS

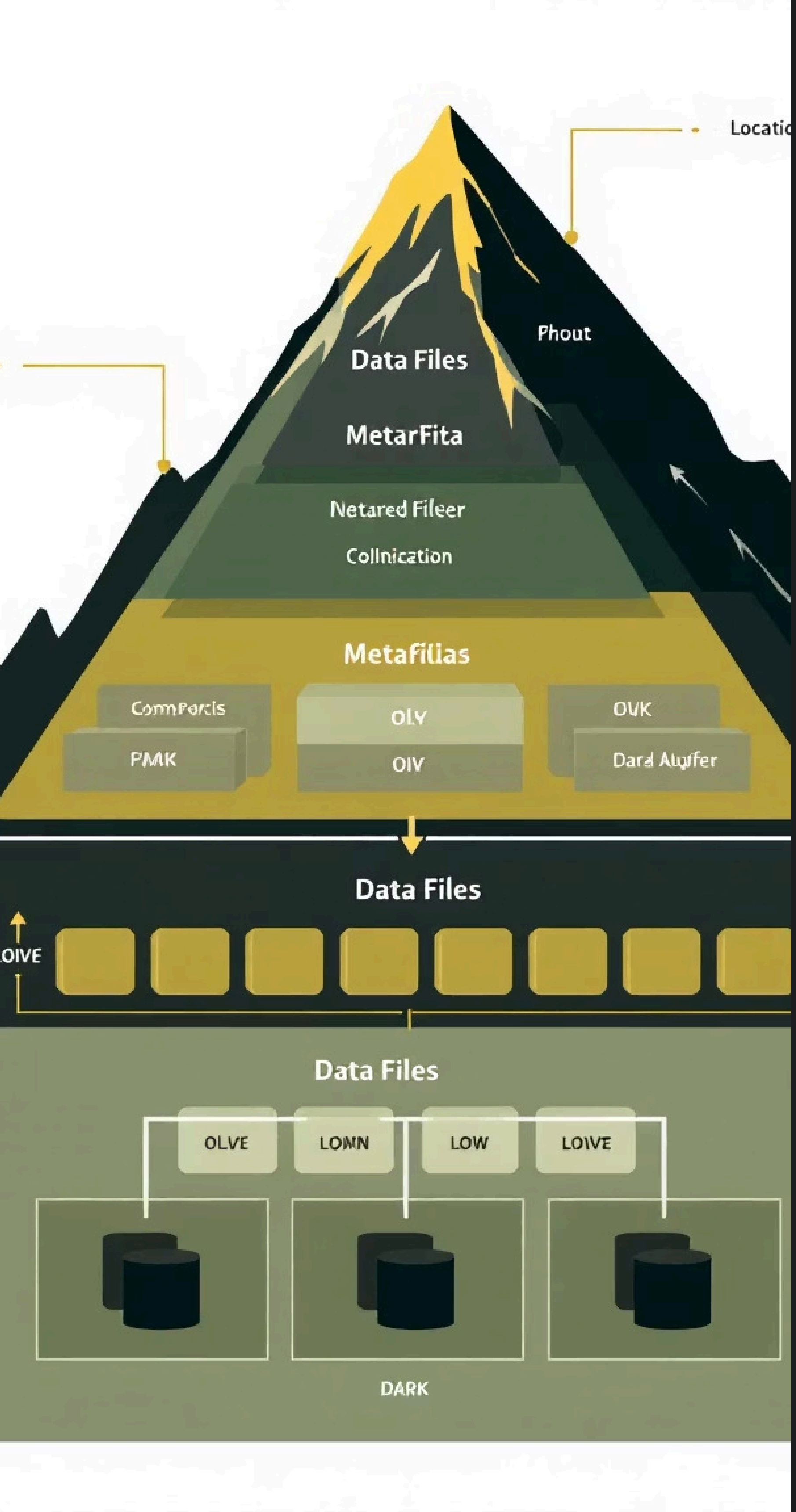
- A table format specification
  - **File structures**: How data files should be organized
  - **Metadata formats**: What information to track and how to store it
  - **File naming conventions**: How files should be named
  - **Data types and schemas**: How to represent different data types
  - **API contracts**: How applications should interact with the data
- A collection of APIs and libraries
- A framework for data management in data lakes
- An interface for applications to interact with table data


Apache Iceberg
Format Specification
Apache Iceberg
Table Fuut Companions
Table Format Specification
(Pabant Ferronment
VFS
Table Fection
Apache Ioolcation
Olive Can Dlechy Laure Olive, Laure Green Stage Green Popae Green,
Table Forma Specification
Apache Format
Ontion Formats
Tablle Scuic Dreen
Table Green
Table Format Specification
Click Back Olive Olive, Lauare Greemy Saage Green
Table For with Inticative Green
© apache iceberg

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

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



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. (Storage trick!)

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.

ACID Transations

ACID = Four Important Guarantees

A = Atomicity (All-or-Nothing)

Example: You want to add 1000 new customers
✅ ALL 1000 customers are added
❌ NO customers are added at all
🚫 NEVER: 347 customers added and then crash

C = Consistency (Always Valid)

Example: You have a rule "age must be > 0"
✅ For every change: all ages remain > 0
❌ Iceberg will NEVER accept negative ages

I = Isolation (No Interference)

Scenario: Two people working simultaneously
Person A: Counts all customers (takes 5 minutes)
Person B: Adds 100 new customers (during those 5 minutes)

✅ Person A sees: The count from 5 minutes ago
✅ Person B: Can continue working normally
🚫 Person A does NOT suddenly see extra customers appearing

D = Durability (Saved Forever)

✅ Once Iceberg says "Saved!", it's REALLY saved
✅ Even if the server crashes immediately after
✅ Even if power goes out
✅ Your data is safe

Why is this special?

Traditional data lakes (without Iceberg):

❌ Person A reads data while Person B writes = chaos
❌ Half-written files when server crashes
❌ Inconsistent data
❌ Lots of manual work to fix problems

With Iceberg:

✅ Works just like a normal database
✅ Multiple people can safely work simultaneously
✅ No data loss
✅ Always consistent results

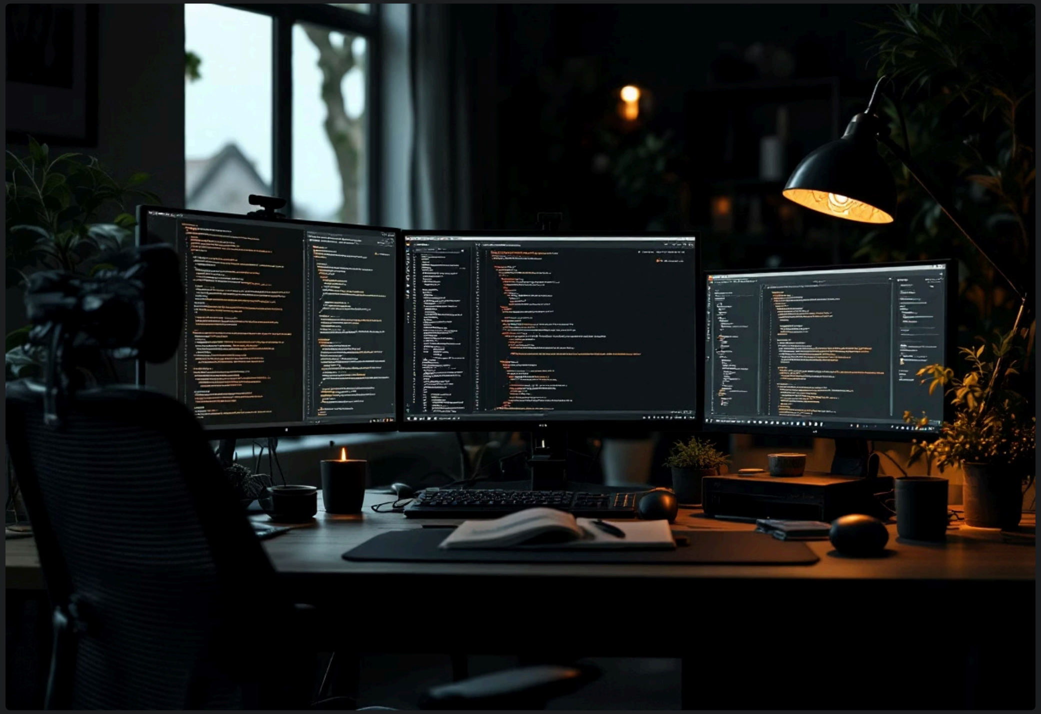
Getting Started: Setting Up Your Environment for Iceberg

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-
spark-runtime-3.5_2.12:1.5.2
```

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")
```

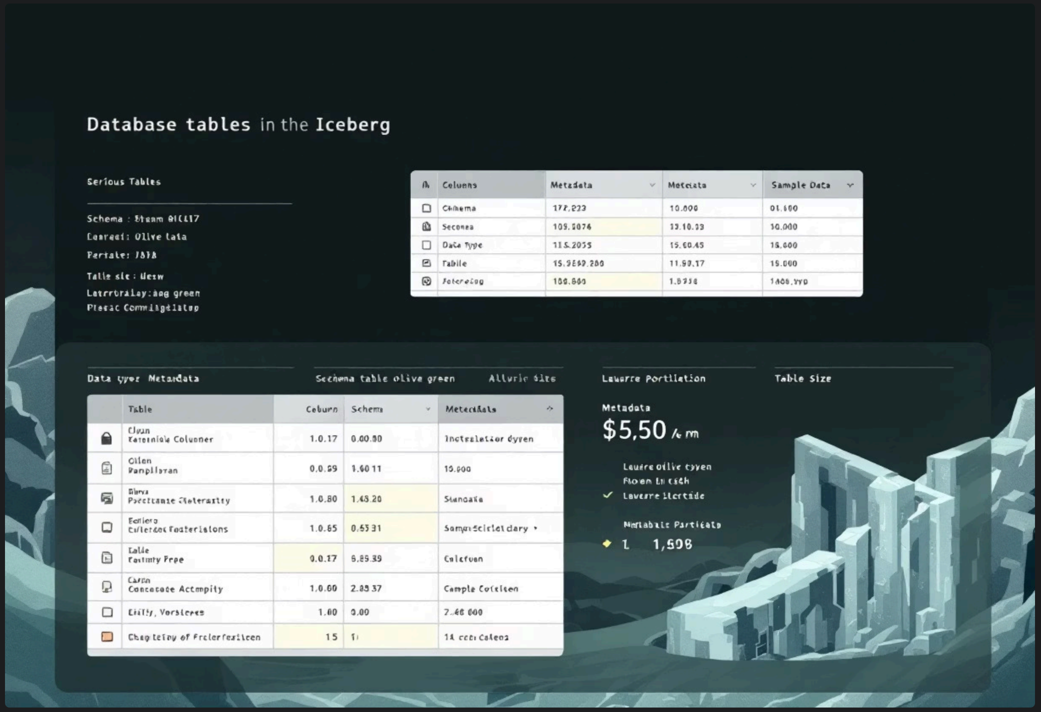
Creating and Managing Iceberg Tables with PySpark

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.custome
rs")
```

Update

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

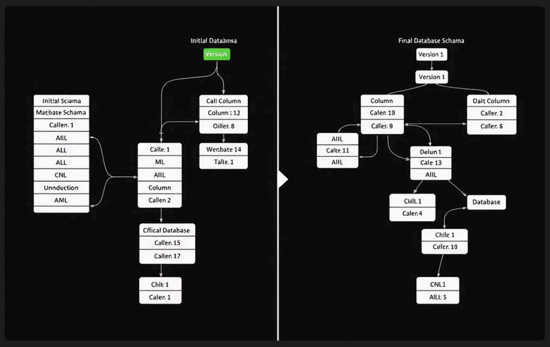
Delete

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

Advanced Features: Time Travel, Schema Evolution and Partitioning

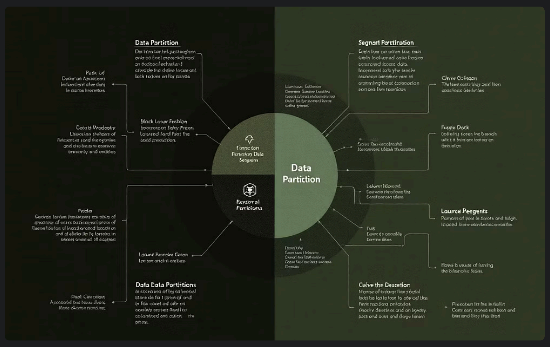
Time Travel

```
# 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: Data Bucketing

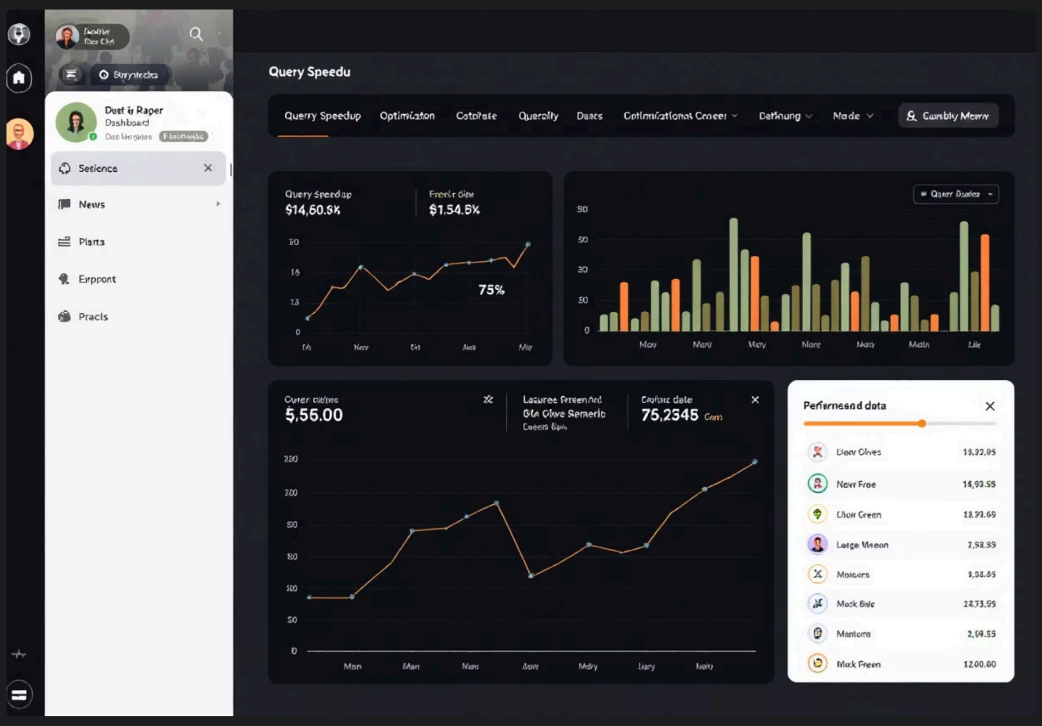
Data bucketing enhances query performance by distributing rows into a fixed number of buckets, determined by the hash of a chosen column. This technique significantly reduces the volume of data scanned during queries involving joins or filters, as Iceberg can intelligently prune irrelevant buckets. It's particularly effective for high-cardinality columns.

```
CREATE TABLE iceberg_db.products (
  product_id INT,
  product_name STRING,
  category STRING,
  price DECIMAL(10, 2)
) USING iceberg
PARTITIONED BY (bucket(16, product_id))

# bucket = hash % 16
```

Performance Optimisation Techniques with Iceberg and PySpark

Data File Optimisations

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



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

Query Optimisations

Partition Pruning

✅ Skip all dates except 2024-06-25
✅ Skip all regions except Europe
Result: 99% of files skipped

Metadata Filtering

From remaining files:
✅ Skip files where max_order_amount < 500
✅ Go directly to files where min_order_amount > 500
Result: Another 80% of files skipped

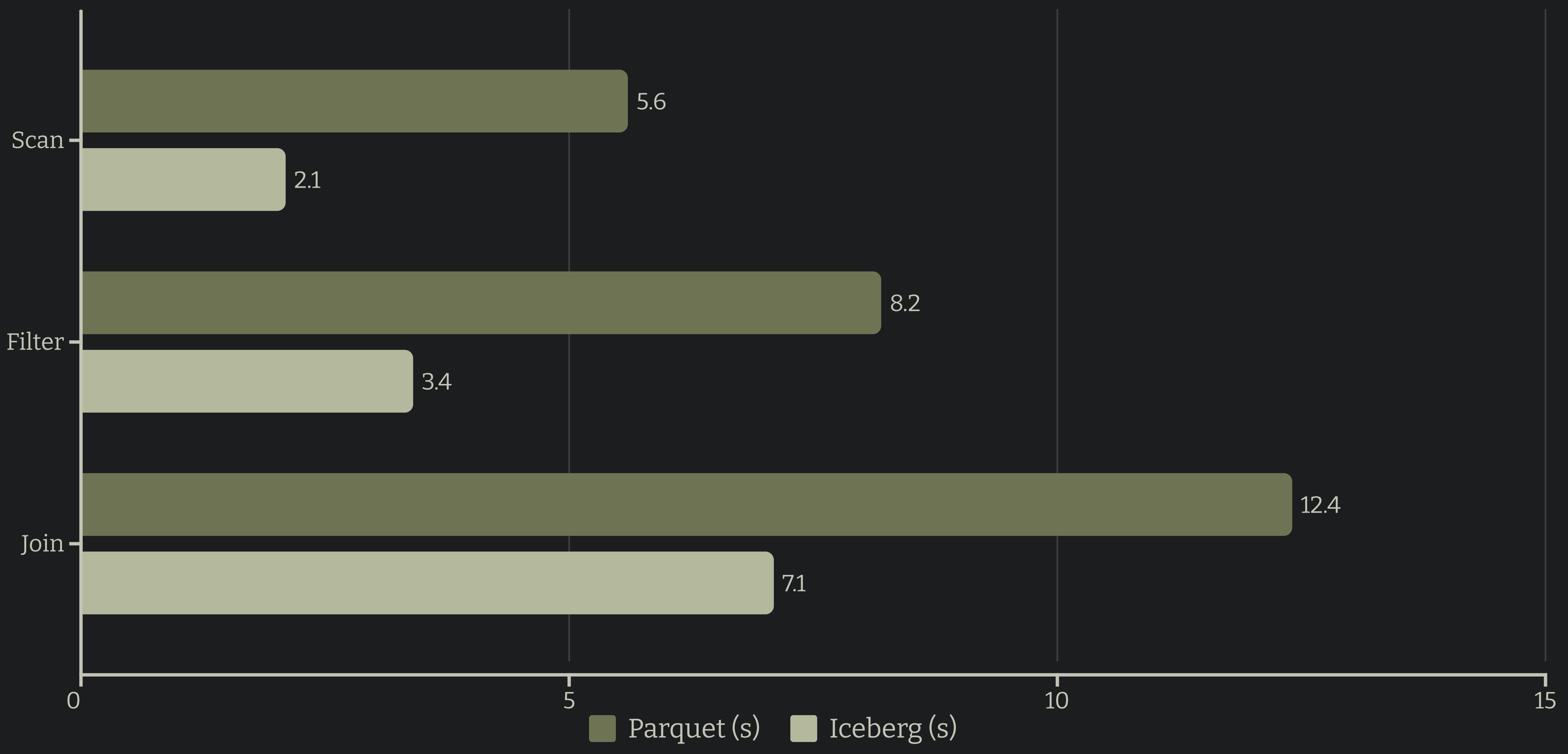
Column Projection

From final files:
✅ Read only customer_id, first_name, last_name, order_amount columns
✅ Skip all other 46 columns
Result: 90% less data transfer

Final result:

From 10TB database → Reads only 50MB
From 2 hour query time → Done in 15 seconds
From €100 cloud costs → Only €0.50

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.