J-FALL 2026 · NEW & COOL

Analytical Java with DuckDB



An in-process analytical database, one JDBC driver away.

Geertjan Wielenga

Java Champion · DuckDB Developer Relations and Community Manager

DuckDB is already in a lot of Java builds



org.duckdb on Maven Central, September to mid-November 2026: 2.1M downloads from 93K unique sources at 8.4K companies. November is a partial month.

Your service just needs a few answers

01

Top products this month

GROUP BY product ORDER BY revenue DESC LIMIT 10

02

p95 latency per region

quantile_cont(latency_ms, 0.95)

03

A CSV export joined with a Parquet dump

'export.csv' JOIN 'dump.parquet'

Not a data platform. Just a handful of analytical questions, asked from inside a Java service.

The usual answers each cost something

THE USUAL ANSWER	WHAT IT COSTS YOU
Run GROUP BYs on the transactional database	A row store doing scans it wasn’t built for, competing with production traffic
Ship the data to a warehouse	A new pipeline, new infrastructure, and answers that arrive later
Hand-write it with Java streams	Every new question is new code: grouping, joins and percentiles by hand

OLTP and OLAP want different shapes

Row store · OLTP



Fetch or update one order with all its fields. Postgres and MySQL are built for exactly this.

Column store · OLAP



Aggregate one field across millions of rows. Read only that column, in big batches.

What if the analytics engine ran inside your JVM?

DuckDB in one slide

In-process

A library inside your JVM, like SQLite. No server, no network hop, no ports.

Columnar

Stores and scans data column by column, so analytical queries read only what they need.

Vectorized

Executes over batches of values instead of one row at a time, and uses all your cores.

Zero infrastructure

One JDBC JAR. In memory or a single file. Free and open source under the MIT license.

Born at CWI in Amsterdam

1990s–2000s

Column stores at CWI

MonetDB, then vectorized execution with X100.

2018–2021

SQLite for analytics

Prototype in 2018, 0.1 at SIGMOD 2019, spin-off in 2021.

2024–2026

1.0, then 2.0 LTS

Stable storage format in June 2024. Quack server in October 2026.

Created by Mark Raasveldt and Hannes Mühleisen. The non-profit DuckDB Foundation keeps it MIT-licensed.

Where DuckDB sits

	SQLITE	DUCKDB	SPARK	CLOUD WAREHOUSE
Runs as	Library in your process	Library in your process	A cluster	A managed service
Built for	Transactions	Analytics	Distributed analytics	Shared analytics at scale
Storage	Row-based file	Columnar file, memory, or your files	Files across many nodes	Managed columnar storage
New ops	None	None	Cluster to run	Accounts, loading, billing

One line to query Parquet on the web

```
FROM 'https://duckdb.org/data/holdings.parquet';

DESCRIBE FROM 'https://duckdb.org/data/holdings.parquet';
SUMMARIZE FROM 'https://duckdb.org/data/holdings.parquet';
SELECT count(*) FROM 'https://duckdb.org/data/holdings.parquet';
FROM parquet_metadata('https://duckdb.org/data/prices.parquet');

FROM 's3://my-bucket/orders/*.parquet';
FROM 'hf://datasets/<org>/<dataset>/**/*.parquet';

CREATE TABLE holdings AS
    FROM 'https://duckdb.org/data/holdings.parquet';
```

One line

A URL is a table; httpfs autoloads on first use.

Range requests

Only the footer and the columns used are fetched.

Any object store

s3://, gs://, az:// and hf://, with globs.

Demo

03

Let’s see code

01 Add the JDBC driver

02 Query Parquet, CSV and JSON files

03 Join files with in-memory data

04 Append Java objects

05 Map results to records

06 Hand off to Apache Arrow

Step 1 · One dependency, one URL

```
<dependency>
    <groupId>org.duckdb</groupId>
    <artifactId>duckdb_jdbc</artifactId>
    <version>[latest]</version>
</dependency>
```

```
// in memory
Connection conn = DriverManager.getConnection("jdbc:duckdb:");

// or persisted to a single file
Connection conn = DriverManager.getConnection("jdbc:duckdb:analytics.duckdb");
```

Step 2 · Query files where they live

```
SELECT product, sum(amount) AS revenue
FROM 'orders/*.parquet'
WHERE order_date >= date_trunc('month', current_date)
GROUP BY product
ORDER BY revenue DESC
LIMIT 10;

SELECT * FROM read_csv('customers.csv');
SELECT * FROM read_json('events.json');
```

Globs welcome

One query over a whole folder of files.

Schemas detected

CSV types are sniffed, nested JSON becomes structs.

Reads less

Parquet: only the columns and row groups needed.

Step 3 · Join files with in-memory data

```
try (Statement stmt = conn.createStatement()) {
    stmt.execute("CREATE TABLE customers AS FROM 'customers.csv'");

    ResultSet rs = stmt.executeQuery("""
        SELECT c.region,
               quantile_cont(r.latency_ms, 0.95) AS p95
        FROM 'requests/*.parquet' r
        JOIN customers c USING (customer_id)
        GROUP BY c.region
        ORDER BY p95 DESC
        """);
}
```

Files, tables and query results all mix in one SQL statement.

Step 4 · Append Java objects in bulk

```
record Request(String customerId, String path, double latencyMs) {}

DuckDBConnection duck = (DuckDBConnection) conn;
try (var appender = duck.createAppender(DuckDBConnection.DEFAULT_SCHEMA, "requests")) {
    for (Request r : batch) {
        appender.beginRow();
        appender.append(r.customerId());
        appender.append(r.path());
        appender.append(r.latencyMs());
        appender.endRow();
    }
}
```

The Appender skips per-row SQL parsing: the fast path for data already in your heap.

Step 5 · Results straight into records

```
record RegionLatency(String region, double p95) {}

List<RegionLatency> result = new ArrayList<>();
try (ResultSet rs = stmt.executeQuery(P95_BY_REGION)) {
    while (rs.next()) {
        result.add(new RegionLatency(rs.getString("region"), rs.getDouble("p95")));
    }
}
```

It’s plain JDBC, so JdbcTemplate, jOOQ, JDBI and friends work as usual.

Step 6 · Columnar hand-off with Apache Arrow

```
// out: query results as Arrow batches
try (var allocator = new RootAllocator();
     var rs = (DuckDBResultSet) stmt.executeQuery("FROM 'requests/*.parquet'");
     var reader = (ArrowReader) rs.arrowExportStream(allocator, 4096)) {
    while (reader.loadNextBatch()) {
        VectorSchemaRoot batch = reader.getVectorSchemaRoot();
    }
}
```

```
// in: query an Arrow stream as if it were a table
duck.registerArrowStream("incoming", arrowStream);
stmt.executeQuery("SELECT count(*) FROM incoming");
```

Real-world patterns

Embedded analytics endpoints

A Spring Boot service answers dashboard queries from Parquet, with no warehouse round-trip.

Fast local ETL jobs

CSV in, cleaned and partitioned Parquet out, in one COPY statement.

SQL over test fixtures

Keep expected data as files and assert with a query instead of hand-built object graphs.

Reading from S3

The httpfs extension queries Parquet straight from object storage.

An analytics endpoint in Spring Boot

```
@RestController
class LatencyController {
    private final JdbcTemplate duck;   // DataSource on jdbc:duckdb:

    @GetMapping("/latency/p95")
    List<RegionLatency> p95() {
        return duck.query("""
            SELECT region, quantile_cont(latency_ms, 0.95) AS p95
            FROM 's3://telemetry/requests/*.parquet'
            GROUP BY region
            """, (rs, i) -> new RegionLatency(rs.getString(1), rs.getDouble(2)));
    }
}
```

Local ETL to S3 in one statement

```
INSTALL httpfs;
LOAD httpfs;
CREATE SECRET (TYPE s3, PROVIDER credential_chain);

COPY (
    SELECT *, strftime(order_date, '%Y-%m') AS order_month
    FROM read_csv('exports/*.csv')
    WHERE status = 'shipped'
) TO 's3://lake/orders' (FORMAT parquet, PARTITION_BY (order_month));
```

Run it from a main method, a scheduled job or a CLI. No cluster required.

SQL over test fixtures

```
@Test
void topProductsMatchFixture() throws SQLException {
    ResultSet rs = stmt.executeQuery("""
        FROM 'src/test/resources/expected-top10.csv'
        EXCEPT
        SELECT product, revenue FROM top_products
        """);
    assertFalse(rs.next(), "expected rows missing from result");
}
```

Fixtures stay readable files; the comparison is one query.

Limits and gotchas

Concurrency

One process owns a database file. Since 2.0, other processes can read and write through it with the Quack server. Many threads in one process are fine.

Memory

Native memory, outside your heap and -Xmx. By default it may use most of the machine's RAM: set memory_limit, and temp_directory so big queries spill to disk.

Single node

Great up to one machine's worth of data and cores. Not a replacement for high-rate transactional writes or a shared, governed store.

When you still need a warehouse

USE DUCKDB WHEN	USE SOMETHING ELSE WHEN
The data fits on one machine, disk included	The data or the queries outgrow one machine
Questions are read-heavy and ad hoc	The workload is many small transactional writes
One service or job owns the data	Many teams share a store that outgrows one server
You want no new infrastructure	You need central access control and governance

THANK YOU

Questions?

DuckDB
duckdb.org

Friends of OpenJDK
foojay.io

Geertjan Wielenga
geertjan@ducklabs.com