GizmoEdge

A Distributed SQL Engine for IoT and Edge Analytics

Philip Moore

GizmoData

DuckDB Developer Meeting #1

January 30, 2026 • Pakhuis de Zwijger, Amsterdam

About Me

Philip Moore

Founder, GizmoData

Data Enthusiast and builder

- Worked with Oracle technology for 15 years
- Expanded to open-source data technologies about 10 years ago
- I love the COUNT DISTINCT problem!
- Worked for P&G, HP, Dunnhumby, Kroger, Voltron Data, and now GizmoData ☺

Background

- Married to Scharlene – 3 kids, 3 dogs, 2 cats
- Former U.S. Marine

Awards

- Marine of the Year (2000) – HQ Battalion 4th MarDiv
- Oracle DBA of the Year – North America 2016

DuckDB Contributions

- **struct_insert() function**
  `PR #3853 – new scalar function for structs`
- **BIT_COUNT HUGEINT support**
  `PR #4440 – extended to 128-bit integers`
- **information_schema views**
  `PR #12942 – referential constraint views`
- **httpfs: S3 Requester Pays mode**
  `PRs #85, #99 – feature + bugfix for session tokens`
- **Documentation contributions**
  `duckdb-web PRs #296, #303, #403, #5581`
- **Issues & feature requests**
  `Bitmap aggregation (#3943), hash distribution (#4417), hive partitioning (#12921), support for encryption of DuckDB database file (password protected) (#4512)`

The Challenge: Data at the Edge

Volume

IoT sensors, mobile devices, and edge nodes generate massive data volumes—faster than they can be centralized.

Cost

Cloud egress, storage, and compute costs scale linearly. Centralizing everything is economically unsustainable.

Latency

Real-time decisions can’t wait for round-trips to the cloud. Analytics must happen where the data lives.

What is GizmoEdge?

A distributed OLAP engine that uses DuckDB as its execution engine—coordinating parallel query execution across heterogeneous workers, from cloud Kubernetes clusters to laptops, Linux boxes, and iOS devices.

⚡ Divide & Conquer

Shards data across workers.
Aggregation queries run in parallel;
results are combined on the server.

📦 DuckDB-Powered

Each worker runs DuckDB locally.
Full SQL support, vectorized execution,
zero external dependencies.

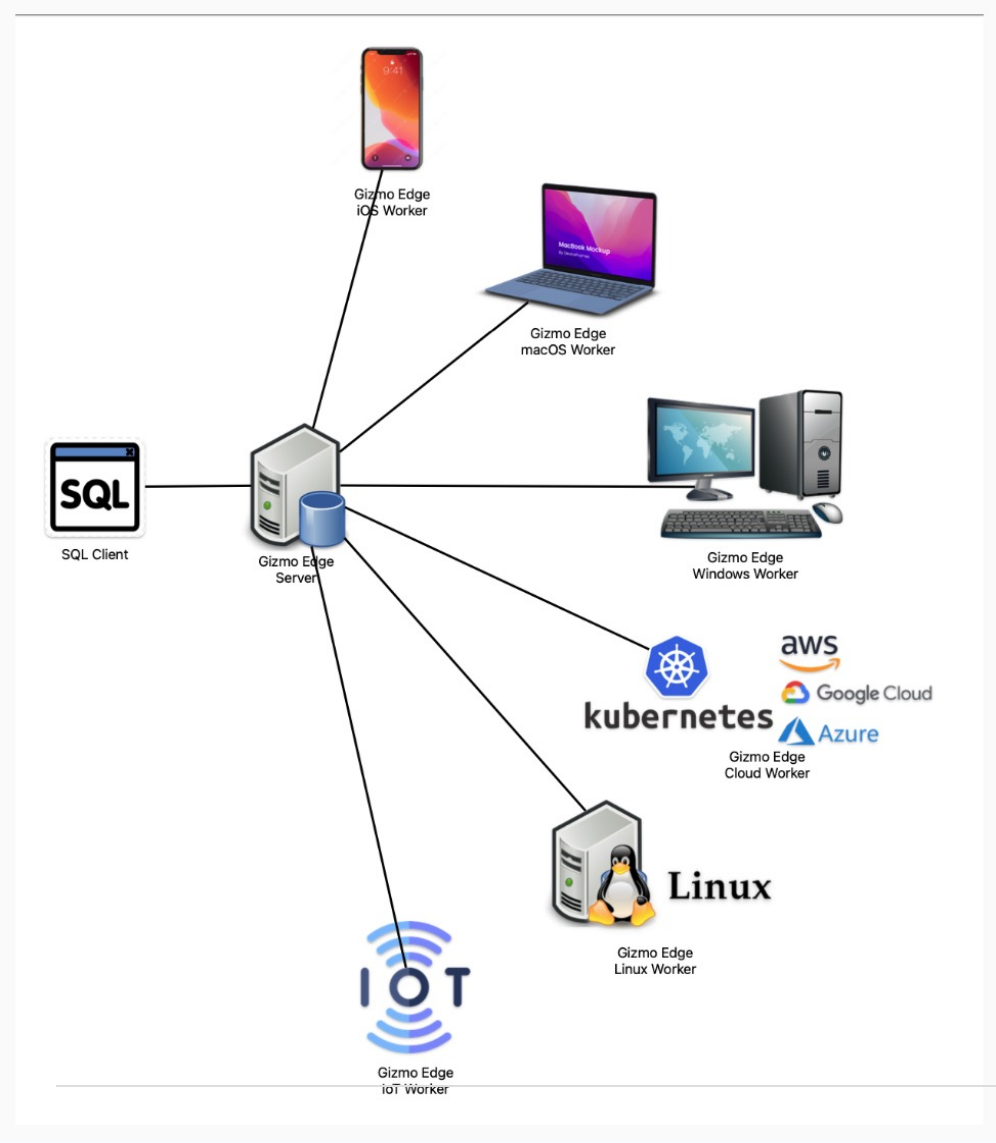
🌐 Heterogeneous Workers

Cloud (K8s on AWS/Azure/GCP),
bare-metal Linux, macOS laptops,
and iOS devices—all as workers.

🔌 Arrow + WebSockets

Apache Arrow IPC for columnar
serialization. WebSocket/TLS for
secure, async communication.

Architecture Overview



Server (Coordinator)

- Parses SQL with PostgreSQL parser (pglast)
- Detects aggregate functions (SUM, AVG, COUNT, MIN, MAX)
- Distributes query to workers with assigned shards
- Aggregates partial results into final answer

Workers

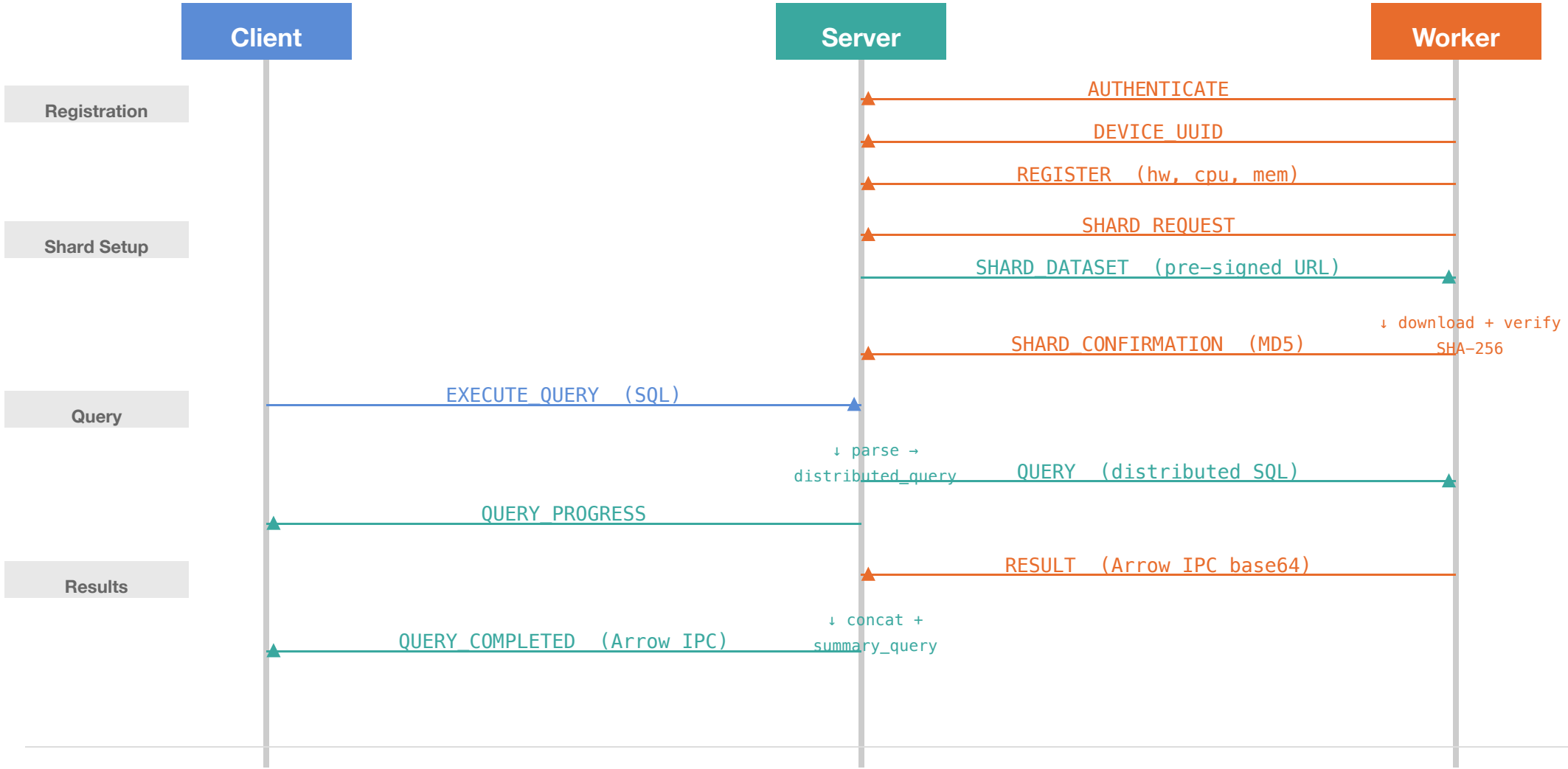
- Download shard via pre-signed S3/Azure URL
- Execute query locally with DuckDB (read-only)
- Return results as Arrow IPC over WebSocket
- Can run on any platform: K8s, Linux, macOS, iOS

Client

- Interactive SQL REPL or web-based SQL Navigator
- Supports .set distributed / .set summarize toggles
- Receives Arrow-serialized results from server

WebSocket Protocol: Message Flow

All communication uses async WebSockets (with optional TLS/mTLS). Messages are JSON with a MessageKind discriminator.



How GizmoEdge Uses DuckDB

On Every Worker

```
# Worker opens shard as read-only DuckDB
con = duckdb.connect(database=shard_file,
                     read_only=True)
con.execute(f"PRAGMA threads={cpu_count}")
con.execute(f"PRAGMA memory_limit='{mem}b'")

# Execute distributed query, return Arrow
result = con.execute(query).fetch_record_batch()
arrow_bytes = get_dataframe_ipc_bytes(result)
```

Server-Side Aggregation

```
# Combine worker results with in-memory DuckDB
combined = pyarrow.concat_tables(worker_tables)

con = duckdb.connect(database=':memory:')
con.execute(f"PRAGMA threads={threads}")

# Run summary query on combined results
final = con.execute(summary_query)
            .fetch_record_batch()
```

Key Design Decisions

- DuckDB's embedded nature = zero deployment friction on edge devices (no server process needed)
- read_only=True on workers — safe concurrent access to shard files
- Container-aware: auto-detects CPU/memory from cgroups for Kubernetes pods
- TPC-H dbgen() built in — used for benchmarking and integration tests

Query Distribution: Divide & Conquer

The PostgreSQL parser (pglast) analyzes the AST to determine if a query contains aggregates. If so, the query is distributed to workers and results are combined.

1. Client Query (TPC-H Q1)

```
SELECT l_returnflag, l_linestatus,
       sum(l_quantity)      AS sum_qty,
       avg(l_quantity)      AS avg_qty,
       avg(l_extendedprice) AS avg_price,
       count(*)             AS count_order
FROM lineitem
WHERE l_shipdate <= '1998-09-02'
GROUP BY l_returnflag, l_linestatus;
```

2. Distributed Query (sent to workers)

```
SELECT l_returnflag, l_linestatus,
       sum(l_quantity)        AS sum_qty,
       SUM(l_quantity)        AS _avg_sum_avg_qty,
       COUNT(l_quantity)      AS _avg_count_avg_qty,
       SUM(l_extendedprice)   AS _avg_sum_avg_price,
       COUNT(l_extendedprice) AS _avg_count_avg_price,
       count(*)               AS count_order
FROM lineitem WHERE ...  GROUP BY ...;
```

⬇️ Results from N workers combined via pyarrow.concat_tables()

3. Summary Query (server-side aggregation)

```
SELECT l_returnflag, l_linestatus,
       SUM(sum_qty)                                    AS sum_qty,
       SUM(_avg_sum_avg_qty) / SUM(_avg_count_avg_qty) AS avg_qty,
       SUM(_avg_sum_avg_price) / SUM(_avg_count_avg_price) AS avg_price,
       SUM(count_order)                                AS count_order
FROM combined_result  GROUP BY l_returnflag, l_linestatus;
```

The AVG Problem in Distributed Queries

You can't average the averages. AVG is not a distributive aggregate—it must be decomposed.

✖ Naïve: AVG of AVGs (Wrong)

```
Worker 1:  AVG(qty) = 10  (5 rows)
Worker 2:  AVG(qty) = 20  (95 rows)
Naïve:     AVG(10, 20) = 15   ✖ WRONG
```

Correct answer: (5×10 + 95×20) / 100 = 19.5

The naïve approach ignores row counts and over-weights small shards.

✓ GizmoEdge: SUM/COUNT Decomposition

```
Worker 1:  SUM=50,  COUNT=5
Worker 2:  SUM=1900, COUNT=95
Server:    SUM(50+1900) / SUM(5+95)
           = 1950 / 100 = 19.5  ✔
```

AVG(x) → SUM(x) + COUNT(x) on workers

Server computes: ΣSUM / ΣCOUNT

Mathematically correct regardless of shard sizes.

Shard Management & Data Flow

1

Bootstrap

DuckDB dbgen() generates TPC-H
data as Parquet, then shards into
N "databases"

2

Compress & Store

Each shard exported to Parquet,
compressed with Zstandard (.tar.zst),
uploaded to S3 or Azure Blob

3

Manifest

YAML manifest tracks shard ID,
name, SHA-256 hash, MD5 hash,
and file size

4

Worker Download

Server sends pre-signed URL
(5 min expiry). Worker downloads,
verifies SHA-256, reports MD5

```
# Export shard to Parquet
db.execute("""EXPORT DATABASE '{dir}' (
  FORMAT PARQUET, COMPRESSION zstd,
  ROW_GROUP_SIZE 100000)""")

# Compress with Zstandard
cctx = zstandard.ZstdCompressor(level=3)
with zstandard.open(path, "wb", cctx=cctx) as f:
    with tarfile.open(fileobj=f, mode="w") as tar:
        tar.add(database_directory)
```

Shard Creation (Zstandard compression)

```
# Worker downloads via pre-signed URL
shard_file = await copy_database_file(
    source_path=presigned_url,
    target_path=local_data_dir)

# Integrity: SHA-256 verify + MD5 proof-of-work
sha256_hash = hashlib.sha256(data).hexdigest()
assert sha256_hash == server_hash  # verify
md5_hash = hashlib.md5(data).hexdigest()
# send md5 back to server as proof
```

Worker Shard Verification

Targeted Broadcast Sharding

Each worker receives a micro data warehouse — a complete star schema with a fraction of the facts and only the dimension rows needed to satisfy joins.

How It Works

① Hash-Partition Facts

Fact table (e.g. lineitem) is split into N shards using hash partitioning — each shard gets an even fraction of rows.

② Filter Dimensions to Match

For each shard, only include dimension rows (e.g. customer, supplier, part) that are referenced by that shard's fact rows.

③ Worker Gets a Star Schema

Each worker receives a self-contained micro data warehouse. Inner joins between facts and dimensions run entirely on the worker.

✓ Why This Matters

- Workers handle full star-schema joins locally
- No coordinator-side join work required
- Dimension data is minimized per shard — less I/O
- Each shard is a self-contained analytical unit

⚡ Future: Bloom Filter Optimization

Instead of exact FK lookups during shard creation, use a bloom filter on dimension keys. Trades a small number of false positives (extra dimension rows) for substantially faster shard build times.

Technology Stack for Preview

Category	Technology	Role
SQL Engine	`DuckDB 1.4.4`	Embedded OLAP engine on every node
Query Parsing	`pglast 7.11`	PostgreSQL parser for AST analysis
Serialization	`Apache Arrow IPC`	Zero-copy columnar data exchange
Compression	`Zstandard`	Shard storage compression (level 3)
Networking	`websockets + TLS/mTLS`	Async WebSocket with mutual TLS
Cloud Storage	`S3, Azure Blob`	Pre-signed URLs for shard download
Auth	`Basic + OAuth2 (Clerk)`	SHA-256 passwords, JWT/JWKS tokens
Deployment	`Docker + Helm/K8s`	Multi-arch (amd64/arm64) images
Language	`Python 3.13+ / asyncio`	Async server with process pool workers
Mobile	`APNS (iOS)`	Push queries to offline mobile workers

Deployment: Cloud to Edge

Kubernetes (Production)

- Helm charts for server, workers, client, and web UI
- Server: 48 CPU cores, 384 GB RAM
- Workers: 3.8 CPU / 30 GB RAM each, NVMe storage
- Tested at TPC-H SF100 to SF10,000
- Multi-cloud: AWS EKS, Azure AKS, GCP GKE
- Coiled integration for elastic scaling

Edge / Local

- pip install gizmo-edge — runs anywhere Python runs
- Docker multi-arch images (amd64 + arm64)
- Bootstrap command: one-line setup with TPC-H data
- iOS workers via APNS push notifications
- macOS / Linux / Windows workers
- Zero-config DuckDB: auto-tunes threads & memory

One-Command Bootstrap for local development

```
# Bootstrap: creates TLS certs, users, TPC-H data, and 11 shards
$ gizmo-edge-bootstrap --client-username=scott --client-password=tiger \
                       --worker-password=united --tpch-scale-factor=1 --shard-count=11

# Start server, workers, and client
$ gizmo-edge-server &
$ for x in {1..11}; do gizmo-edge-worker --tls-roots=tls/server.crt --password=united & done
$ gizmo-edge-client --tls-roots=tls/server.crt --username=scott --password=tiger
```

Why DuckDB is the Perfect Fit

Embedded, Zero Dependencies

No server process to manage. A single file is the database. Workers open DuckDB databases as read-only—no coordination overhead, no shared state.

Runs Everywhere

From 384 GB Kubernetes pods to a Raspberry Pi. DuckDB auto-tunes to available CPU and memory. GizmoEdge reads cgroup limits for containers.

Built-in TPC-H Generator

DuckDB's dbgen() function generates benchmark data natively. GizmoEdge uses this for bootstrapping test environments and integration tests.

Arrow-Native

DuckDB's fetch_record_batch() returns Arrow record batches directly—no conversion step. This is the data format that flows over the wire.

Rich SQL Support

Full SQL with window functions, CTEs, complex expressions. The PostgreSQL parser ensures GizmoEdge understands the same SQL dialect DuckDB speaks.

Live Demo

Distributed TPC-H Query Execution

1. Azure AKS cluster – 1,000 workers – 4,000 CPUs
2. Start server + multiple workers
3. Run TPC-H Queries with aggregates, joins, and filters
4. Show distributed vs. non-distributed execution
5. Toggle .set summarize = false to see raw worker results

What's Next

1. **Smarter Query Planning**

   Cost-based decisions on distribute vs. server-side execution. Support for more complex aggregation patterns (HAVING, nested aggregates, windowing functions, etc).

2. **Dynamic Worker Discovery**

   Workers self-register and auto-scale. Shard re-balancing when workers join or leave the cluster.

3. **DuckDB Extension Integration**

   Leverage DuckDB extensions (spatial, ICU, httpfs) on workers. Extension-aware shard distribution.

4. **Expression-Level AVG Support**

   Handle AVG over expressions like AVG(price * (1-discount)), not just simple column references.

5. **Production Hardening**

   Proper KDF for authentication (argon2), connection pooling, retry logic, and observability (OpenTelemetry).

GizmoEdge
Distributed SQL Engine

GizmoData

Thank You!

Web	gizmodata.com/gizmoedge
GitHub	github.com/gizmodata/gizmo-edge
Speaker	Philip Moore

Also check out GizmoSQL

Our single-node Arrow Flight SQL server — powered by DuckDB.
gizmodata.com/gizmosql

Questions?