



Stop Torturing PostgreSQL

Why Your PostgreSQL Database
Needs a ClickHouse Sidekick

22.10.2026

Dirk Krautschick

[PGConf.EU](#) Valencia

#whoami



Dirk Krautschick
Senior Solution Architect



since Nov 2023

19 years

**DBA, Trainer, Consulting,
Sales Engineering**

PostgreSQL, Oracle,...

...and Kafka, Clickhouse, OpenSearch,...

Married, 2 Junior DBAs

**mountainbike, swimming,
movies, music,
hifi/homecinema, 3 bit
computing**



PostgreSQL User Group NRW

North Rhine-Westphalia, Germany

Founded Dec 2023

**So far in Cologne, Aachen, Essen,
Paderborn, M'Gladbach, Dortmund,
Cologne (again :-)),...**

~ 3 meetups a year, all around NRW

Next on Nov 26 in M'Gladbach (again :-))

With Credativ

Upcoming events in the pipeline, stay tuned!



Before we start...

Disclaimer

Different audience, different perspectives

**I am NOT a deep-dive analytics guru...
...just a desperate database guy... like you!?**

Open Source

**Everything that looks like advertising won't be meant like such...
...but still could be highly recommended!!! ;-)**

Once upon a time...

...being a PostgreSQL person was simple:

OLTP and OLAP? Just one box, one engine

"PostgreSQL is the everything database" - and it mostly was

But today...

Not just ordinary OLTP clients anymore

Dashboards, BI tools, event analytics, AI pipelines

Data grows faster than your maintenance window

"Everything" still has a limit

PostgreSQL rules for all!!!

Extensibility is powerful, that's for sure!

More indexes (BRIN, GIN,...)

Materialized views, partitioning, parallel workers

Just throw more hardware at it, KIWI

...but then...



You hit the "analytical wall"

SELECT COUNT(*) starts taking minutes

Your indexes eat more disk than your actual data

```
postgres=# EXPLAIN ANALYZE
postgres-#   SELECT country, sum(liters)
postgres-#   FROM beer_sales GROUP BY country;
```

QUERY PLAN

```
-----
HashAggregate  (cost=8231409.12..8231409.44 rows=32)
  Group Key: country
    -> Seq Scan on beer_sales  (rows=2100000000)
Planning Time: 4.912 ms
Execution Time: 214532.880 ms  <-- 3.5 minutes, and counting
```

...but then...

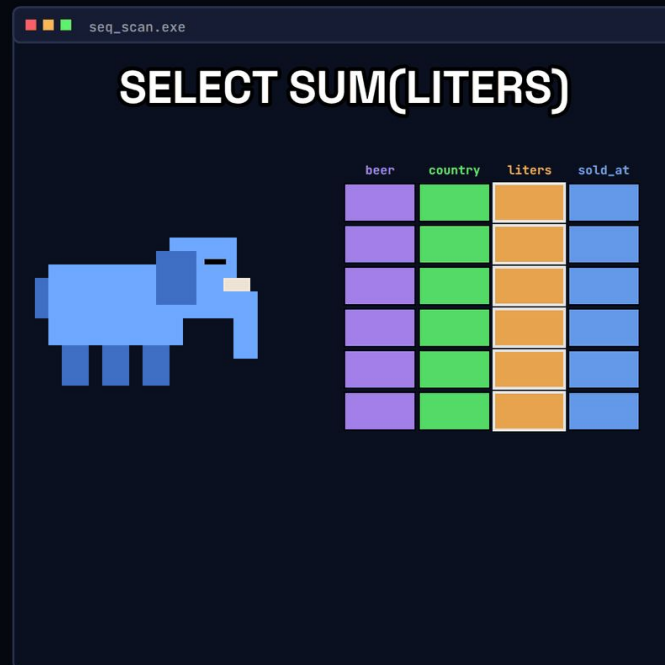
Row-store architecture has limits

Reading a whole row just to sum one column

Aggregations scan far more bytes than they need

More indexes = faster reads, slower writes, more disk

Materialized views help – until refresh itself becomes the bottleneck



The idea

Stay open source. Add a column-store sidekick.

The traditional "solution"

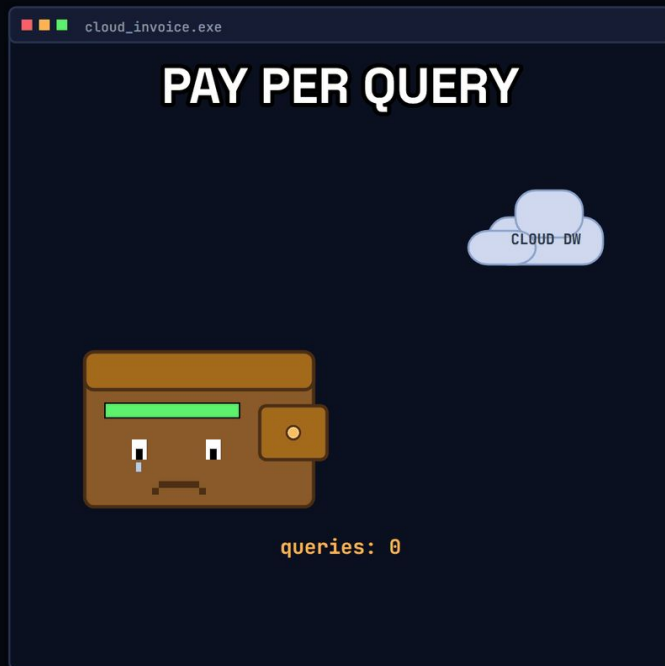
move to a closed-source, pay-per-query cloud warehouse

The better solution

keep PostgreSQL for OLTP, add a Column-Store companion for OLAP

PostgreSQL handles mission-critical writes

The sidekick handles the heavy analytical reads



Let's introduce...



No, not your mouse click history.

And no, not a real-estate platform either



Let's introduce...



...it's not Apache-governed either.

Just Apache-2.0 licensed. Confusing enough?

Good, let's move on



Let's introduce...

ClickHouse



Started as an internal project
at Yandex, 2009

Open sourced June 2016,

Apache License 2.0

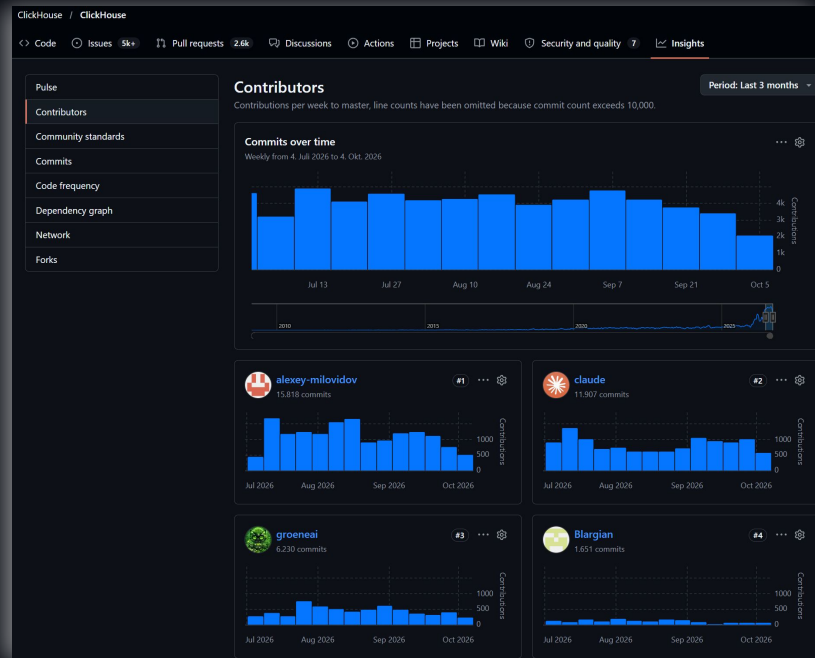
<https://github.com/ClickHouse/ClickHouse>

ClickHouse Inc. founded Sep 2021

(independent company)

Actual stable release v26.8 LTS (Sep 2026)

C++ based



Architecture, side by side

PostgreSQL row-oriented · built for OLTP

Clients

many short transactions



Backend process per connection

postmaster forks one OS process each



shared_buffers

8 kB pages cached in RAM



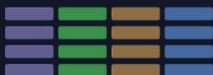
WAL

write-ahead log for durability



Heap + indexes

row tuples stored in 8 kB pages



ClickHouse column-oriented · built for OLAP

Clients

native TCP / HTTP



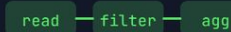
Server: thread pool

one query runs on all cores



Vectorized pipeline

batches of values, not row by row



Sparse primary index

skips whole granules



MergeTree parts

one file per column, background merges



Row-store vs. Column-store

	PostgreSQL (row-store)	ClickHouse (column-store)
Storage	Whole rows together	Each column stored separately
Best at	Many small OLTP reads/writes	Scanning millions of rows for few columns
Aggregations	Reads full rows, filters after	Reads only the columns it needs
Compression	Moderate	Very high - similar values sit together

What gets read?

```
SELECT country, sum(liters) FROM beer_sales GROUP BY country;
```

PostgreSQL

row-store

beer_name	country	liters	sold_at
Pils	DE	0.5	2026-10-01
Stout	IE	0.4	2026-10-01
IPA	US	0.3	2026-10-02
Lager	CZ	0.5	2026-10-02
Weizen	DE	0.5	2026-10-03
Kölsch	DE	0.2	2026-10-03

on disk



bytes read



0 %

ClickHouse

column-store

beer_name.bin	country.bin	liters.bin	sold_at.bin
Pils	DE	0.5	2026-10-01
Stout	IE	0.4	2026-10-01
IPA	US	0.3	2026-10-02
Lager	CZ	0.5	2026-10-02
Weizen	DE	0.5	2026-10-03
Kölsch	DE	0.2	2026-10-03

on disk



bytes read



0 %

Components, side by side

	PostgreSQL	ClickHouse
Processes	process per connection	one server, thread pool
Unit on disk	8 kB page of row tuples	part = dir of column files
Reads	pages via shared_buffers	granules (~8,192 rows) via marks
Writes	WAL, then in-place page change	immutable parts, append-only
Housekeeping	autovacuum, checkpointer	background merges
Index	B-tree: entry per row	sparse: entry per granule

MergeTree in one picture

1 INSERT = new part

background merges combine them

2 A part = a directory

one file per column

3 Sparse index = skip

one key per granule

Writes: batch the inserts, merge later · **Reads:** skip granules, read only the needed columns

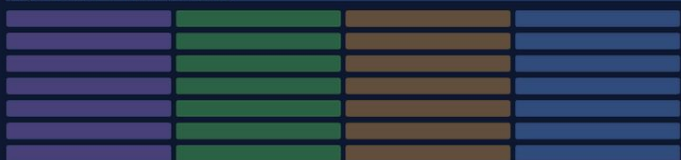
part = directory · granule $\approx$ 8,192 rows · primary.idx = the first key of every granule · illustrative numbers

On disk: from table to granule

PostgreSQL files of 8 kB pages

```
PGDATA/  
| base/<db oid>/  
|   |<relfilenode>  
|   |<relfilenode>.1  
|   |<relfilenode>_fsm  
|   |<relfilenode>_vm  
| pg_wal/
```

page header + item pointers



ClickHouse directories of column files

```
store/xxx/<table uuid>/  
| all_1_4_2/  
|   | columns.txt count.txt checksums.txt  
|   | primary.idx  
|   | country.bin country.mrk2  
|   | liters.bin liters.mrk2  
|   | ... one .bin + .mrk2 per column  
| all_5_5_0/
```

G0

G1

G2

G3

G4

G5

MergeTree & vectorized execution

MergeTree table engine family
background merges of sorted parts

Data physically ordered by a chosen key (ORDER BY)

Vectorized execution
operates on batches of values, not row by row

Parallel, multi-core query execution out of the box

Core concept

MergeTree: parts and merges

sorted by ORDER BY (country, sold_at)

level 0 - one part per INSERT

active parts

4

Core concept

Inside a part

```
SELECT sum(liters) FROM beer_sales WHERE country = 'DE';
```

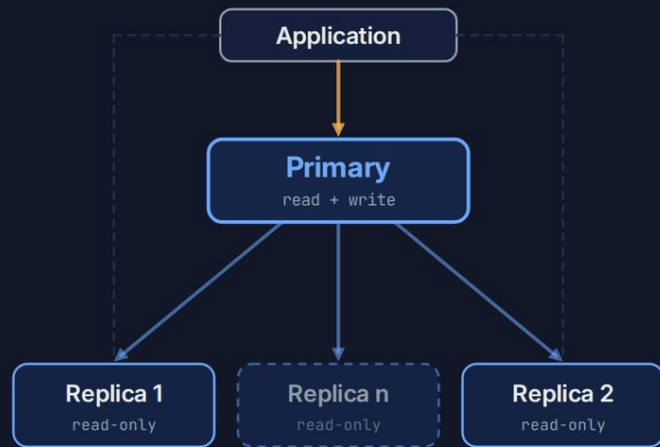
sum(liters)
 Σ

```
part = directory: <col>.bin (compressed data) · <col>.mrk (offsets) · primary.idx · granule ≈ 8 192 rows
```

Scaling out: replicas and shards

PostgreSQL

one primary, many read replicas



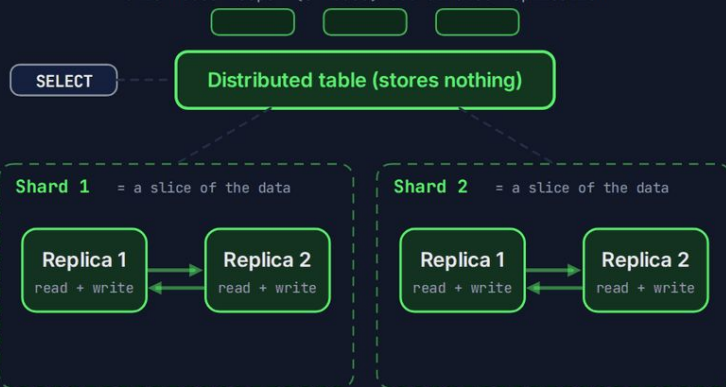
sharding: partitioning + FDW, or an extension like Citus
failover tooling is external (e.g. Patroni)

reads scale with replicas, writes stay on ONE primary

ClickHouse

shards x replicas, every replica writes

ClickHouse Keeper (3 nodes) coordinates replication



INSERT

adding a replica is easy, adding a shard means moving data
insert_quorum = wait for more than one replica

shards split the data, replicas copy it, every replica takes writes

Scaling out, side by side

	PostgreSQL	ClickHouse
Replicas	read-only standbys (streaming)	copies of a shard, all writable
Writes	one primary	any replica, async multi-master
Reads	spread over the replicas	spread over replicas and shards
Sharding	partitioning, FDW, Citus	shards + Distributed table
Adding nodes	clone a standby	replica easy, shard = move data
Consistency	replicas lag (sync optional)	eventual (insert_quorum optional)

The quick binary (testing/dev)... the raw quickstart

One binary, zero dependencies, runs clickhouse-server / -client / -local

Because every DB talk needs a 3-liner install, right?

```
# curl https://clickhouse.com/ | sh
./clickhouse server &
```

```
# in a second terminal
./clickhouse client
```

```
-- Yeah... that's it! Not for production, but great to kick the tires.
```

Production install: Debian and RedHat

Signed packages from the official ClickHouse repository

```
# Debian / Ubuntu
```

```
K=/usr/share/keyrings/clickhouse-keyring.gpg; P=https://packages.clickhouse.com
curl -fsSL $P/rpm/lts/repodata/repomd.xml.key | sudo gpg --dearmor -o $K
echo "deb [signed-by=$K] $P/deb stable main" | sudo tee /etc/apt/sources.list.d/clickhouse.list
sudo apt-get update && sudo apt-get install -y clickhouse-server clickhouse-client
sudo systemctl enable --now clickhouse-server
```

```
# RedHat / CentOS / Fedora
```

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://packages.clickhouse.com/rpm/clickhouse.repo
sudo yum install -y clickhouse-server clickhouse-client
sudo systemctl enable --now clickhouse-server
```

First steps

```
local-host :) SHOW DATABASES;
```

name
INFORMATION_SCHEMA
default
system

```
local-host :) CREATE DATABASE shop;
```

```
local-host :) USE shop;
```

My favourite test data: beer, obviously

```
CREATE TABLE beer_sales
(
    beer_name String,
    country   String,
    liters    Float32,
    sold_at   DateTime
)
ENGINE = MergeTree
ORDER BY (country, sold_at);

INSERT INTO beer_sales VALUES
('Chimay', 'BE', 12.5, now()),
('Leffe bruin', 'BE', 9.0, now()),
('St. Bernadus Abt 12', 'BE', 15.0, now());
```

Hands on

Demo 1



https://github.com/dkrautschick/clickhouse_pg_integration

Nice, but...

What about my actual Postgres data?

The sidekick is useless without a way to feed it.



Every way to get data from A to B

1. The caveman way – manual export/import
2. Query federation – ClickHouse reads Postgres live
- 2b. pg_clickhouse – Postgres queries ClickHouse
3. Refreshable MVs – ClickHouse keeps a copy in sync
4. Kafka CDC – real streaming, row by row
5. The lazy way – Aiven's built-in data service integrations

Path 1 - The caveman way

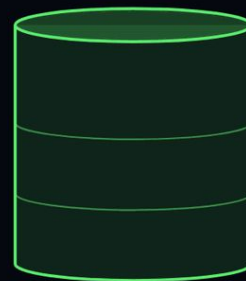
COPY out, COPY in



PostgreSQL



beer_sales.csv



ClickHouse

COPY out, COPY in

Works. Fine for a one-off migration. Terrible for anything live.

```
-- On PostgreSQL
\copy beer_sales TO 'beer_sales.csv' CSV HEADER

-- On ClickHouse
clickhouse-client --query="
  INSERT INTO beer_sales FORMAT CSVWithNames" \
  < beer_sales.csv
```

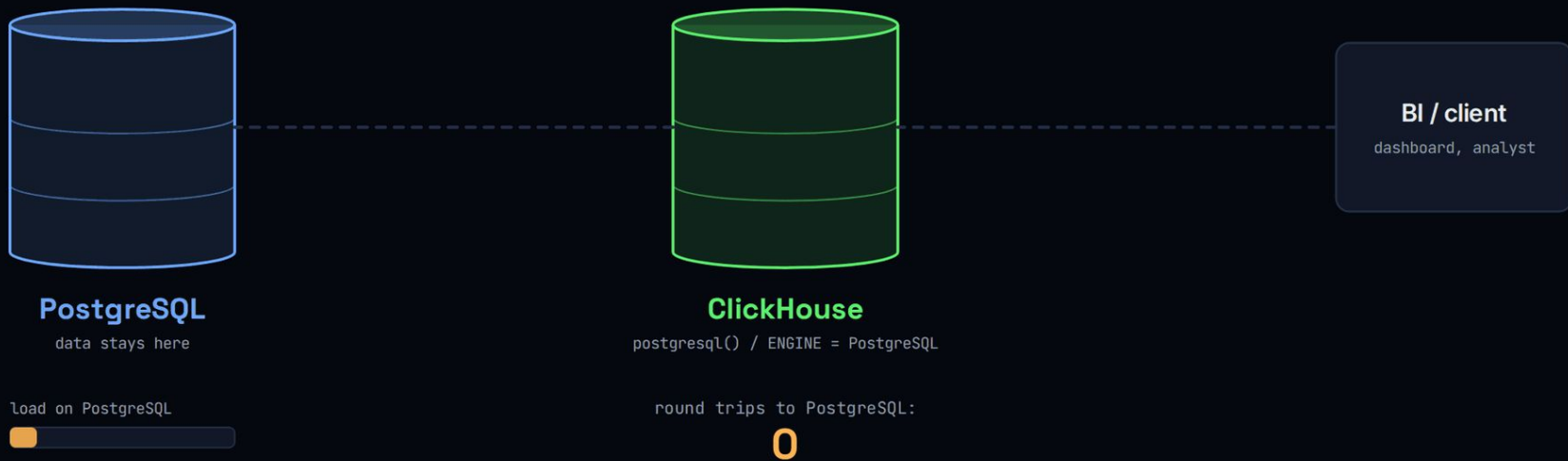
Pro

dead simple

Cons

stale the moment you press Enter.

ClickHouse reads Postgres live



ClickHouse reads Postgres live

The postgresql() table function – or a permanent PostgreSQL table engine

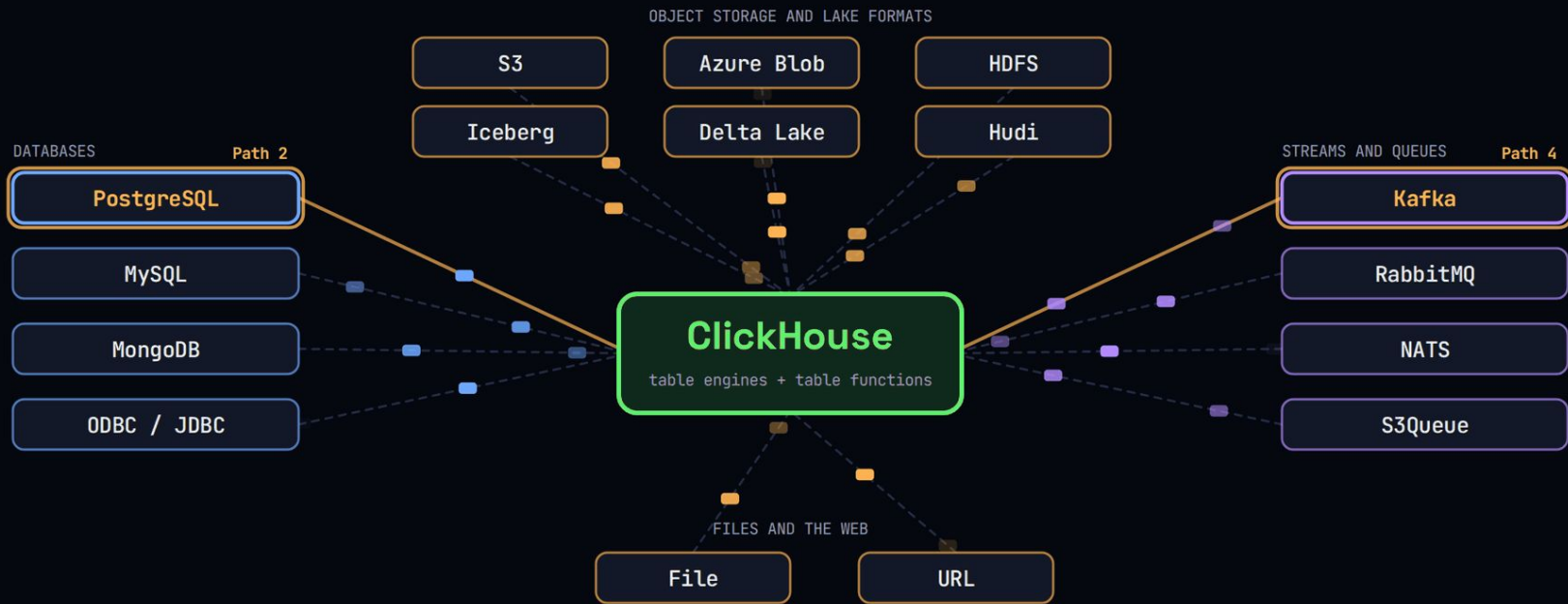
```
-- one-off, ad-hoc
SELECT * FROM postgresql(
    'pg-host:5432', 'shop', 'beer_sales', 'ro_user', 'secret'
);

-- permanent, queryable like any table
CREATE TABLE pg_beer_sales
(
    beer_name String, country String,
    liters Float32, sold_at DateTime
)
ENGINE = PostgreSQL('pg-host:5432', 'shop', 'beer_sales', 'ro_user', 'secret');
```

Every query round-trips to Postgres. Great for exploration, rough at scale.

A quick aside

ClickHouse reads far more than Postgres



Query it live ...

... or copy it into MergeTree with `INSERT ... SELECT`

pg_clickhouse: Postgres queries ClickHouse

Until recently: no official ClickHouse FDW
only a community `clickhouse_fdw`

Now: `pg_clickhouse` (Apache-2.0)
Postgres extension with a ClickHouse FDW
existing queries run unchanged
aggregates + SEMI JOINS pushed down

Needs PG 13+ and ClickHouse 23.3+, TLS
read-focused, DML is on the roadmap

The mirror image of Path 2
Postgres asks, ClickHouse answers



Be careful with the terms!

"Materialized View" means two different things

In PostgreSQL

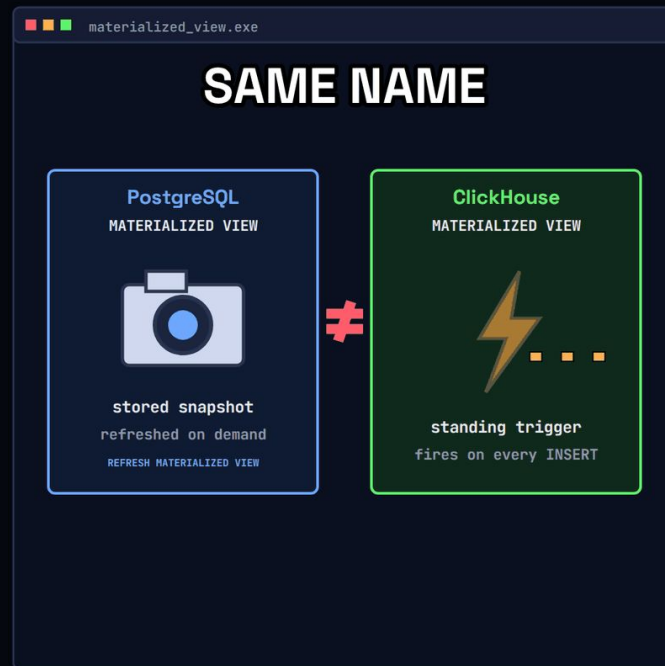
stored snapshot,
refreshed on demand (REFRESH
MATERIALIZED VIEW)

In ClickHouse

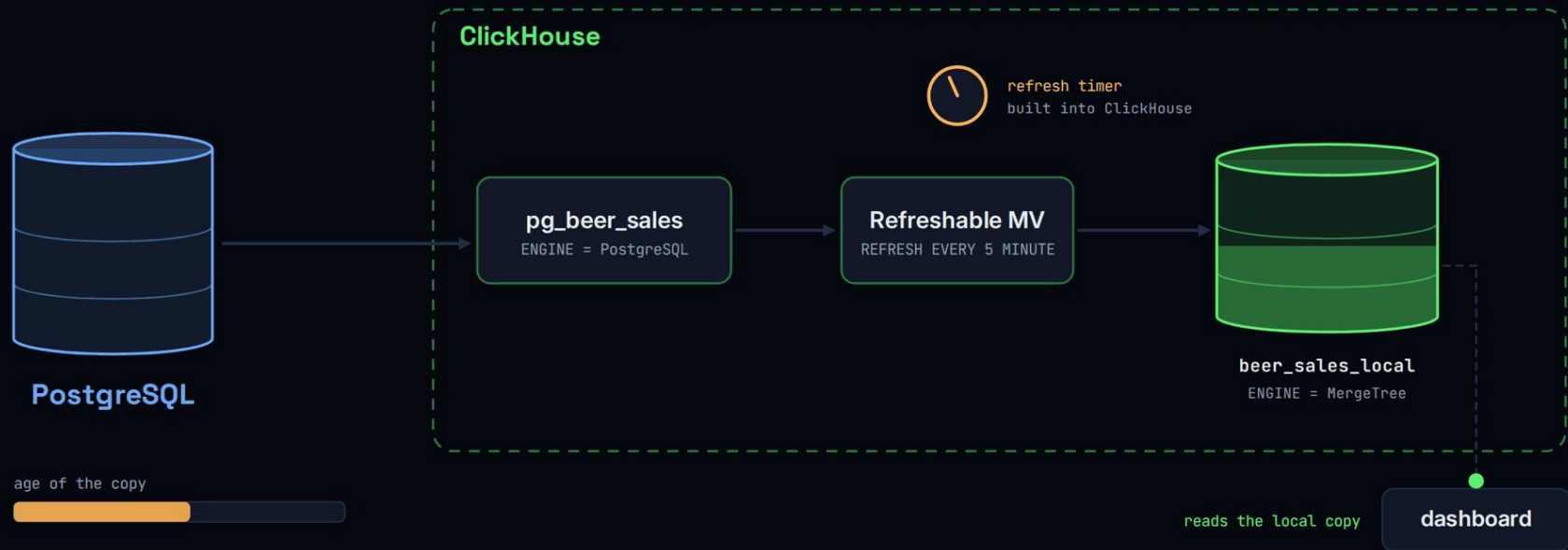
a standing trigger
fires on every INSERT into its source
or a refreshable snapshot: REFRESH EVERY ...

Same words, completely different execution model

This is where the real streaming magic happens



ClickHouse keeps its own copy



Let ClickHouse keep its own copy in sync

```
CREATE TABLE beer_sales_local
(
    beer_name String, country String,
    liters Float32, sold_at DateTime
)
ENGINE = MergeTree ORDER BY (country, sold_at);

CREATE MATERIALIZED VIEW beer_sales_mv
REFRESH EVERY 5 MINUTE
TO beer_sales_local AS
SELECT * FROM pg_beer_sales;
```

ClickHouse schedules the pull itself – no cron, dbt or Airflow needed.

Kafka CDC

If you caught my talk

"Why should Database people care about Kafka and Debezium?"

<https://www.youtube.com/watch?v=0UxOIYS-Te0>

(from PGDay Paris 2025)

this is where it pays off.



Quick Debezium recap

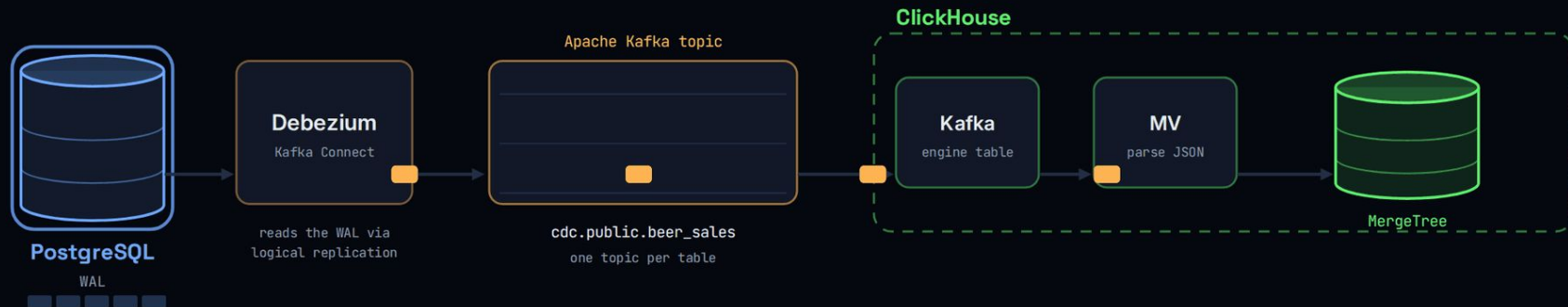
Debezium reads the PostgreSQL WAL via logical replication

wal_level = logical, a replication slot, pgoutput plugin

Every row change becomes an event on a Kafka topic

cdc.public.beer_sales – one topic per table

Real streaming, row by row



```
{"op": "c", "after": {"beer_name": "Pils", "country": "DE", ...}}
```

each INSERT / UPDATE / DELETE becomes one event

ClickHouse consumes the Kafka topic directly

```
CREATE TABLE beer_sales_queue
(
    payload String
)
ENGINE = Kafka
SETTINGS
    kafka_broker_list = 'kafka-dirk.aivencloud.com:13138',
    kafka_topic_list = 'cdc.public.beer_sales',
    kafka_group_name = 'clickhouse-sidekick',
    kafka_format = 'JSONAsString';
```

...and hands it off to a Materialized View

```
CREATE MATERIALIZED VIEW beer_sales_consumer
TO beer_sales_local AS
SELECT
  JSONExtractString(payload, 'after', 'beer_name') AS beer_name,
  JSONExtractString(payload, 'after', 'country')   AS country,
  JSONExtractFloat(payload, 'after', 'liters')      AS liters,
  now()                                             AS sold_at
FROM beer_sales_queue;
```

**Every Debezium event is transformed and pushed into MergeTree
no polling**

Hands on - End to end

Demo 2

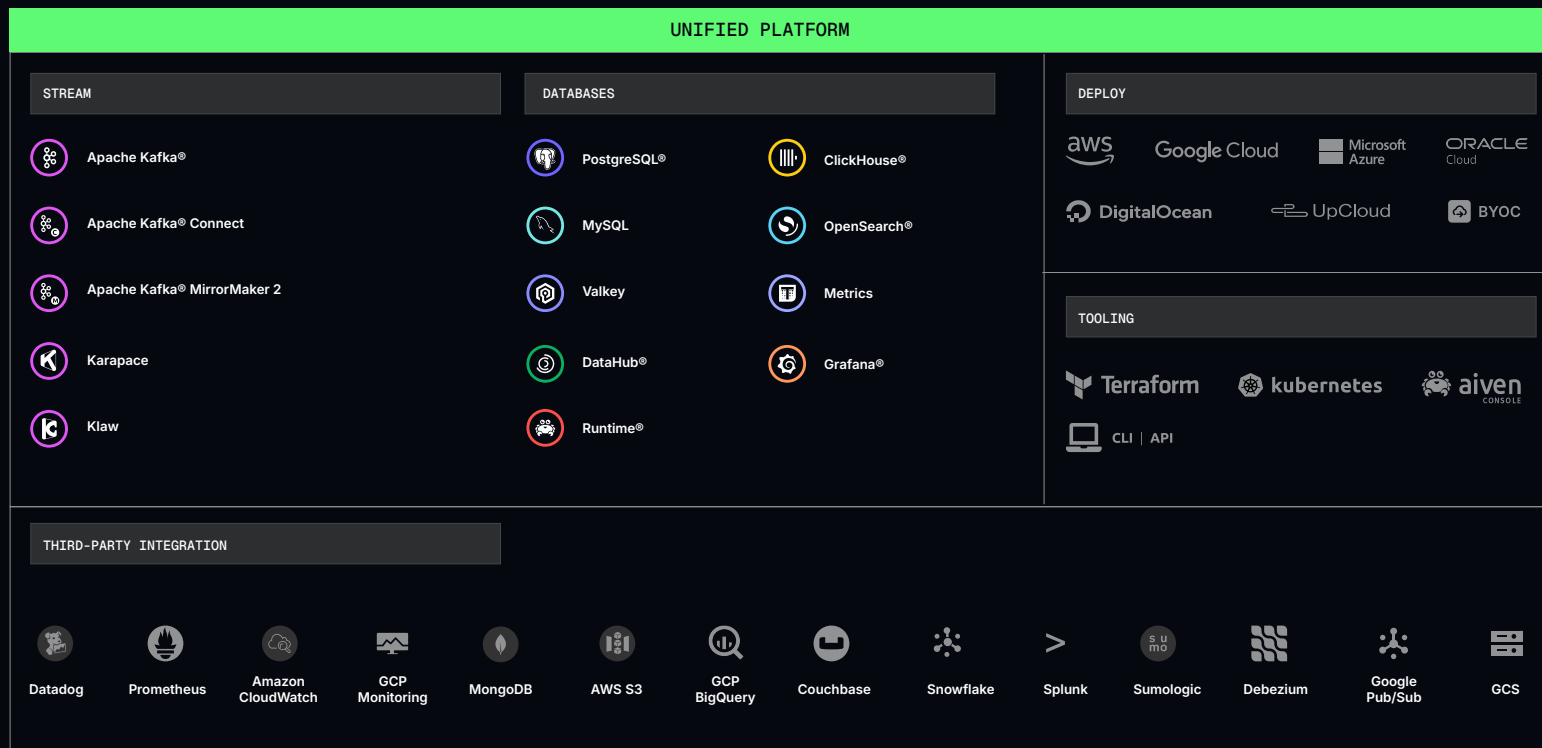


https://github.com/dkrautschick/clickhouse_pg_integration

Let the platform do the wiring

**Aiven for ClickHouse ships both integrations natively
no Debezium config, no Kafka Engine SQL**

AIVEN PLATFORM



Two native integration types, one console click

Integration type	What it wires up
clickhouse_postgresql	Aiven for PostgreSQL → integration database in Aiven for ClickHouse
clickhouse_kafka	Aiven for Apache Kafka → integration database in Aiven for ClickHouse

Let the platform do the wiring



Be lazy using Terraform!

Two services, fully as code

```
resource "aiven_pg" "shop" {  
  project      = var.project  
  service_name = "shop-pg"  
  plan         = "startup-4"  
}  
  
resource "aiven_clickhouse" "sidekick" {  
  project      = var.project  
  service_name = "beer-clickhouse"  
  plan         = "startup-16"  
}
```

Be lazy using Terraform!

...wire both integrations together

```
resource "aiven_service_integration" "pg_to_ch" {
  integration_type      = "clickhouse_postgresql"
  source_service        = aiven_pg.shop.service_name
  destination_service   = aiven_clickhouse.sidekick.service_name
}

resource "aiven_service_integration" "kafka_to_ch" {
  integration_type      = "clickhouse_kafka"
  source_service        = aiven_kafka.events.service_name
  destination_service   = aiven_clickhouse.sidekick.service_name
}

% terraform init && terraform apply -var-file=var-values.tfvars
```

No Debezium connector JSON. No Kafka Engine SQL. Just four resources.

Aiven console walk-through

Demo 3



https://github.com/dkrautschick/clickhouse_pg_integration

Which path, when?

Path	Latency	Ops effort
1. CSV export/import	hours/days	low, but manual every time
2. Query federation	seconds (live)	low, but slow at scale
2b. pg_clickhouse	seconds (live)	low - just an extension
3. Refreshable MV	seconds to minutes	low - ClickHouse refreshes itself
4. Kafka CDC	sub-second	high - Kafka + Debezium to run
5. Aiven integrations	sub-second	low - Aiven runs the pipeline

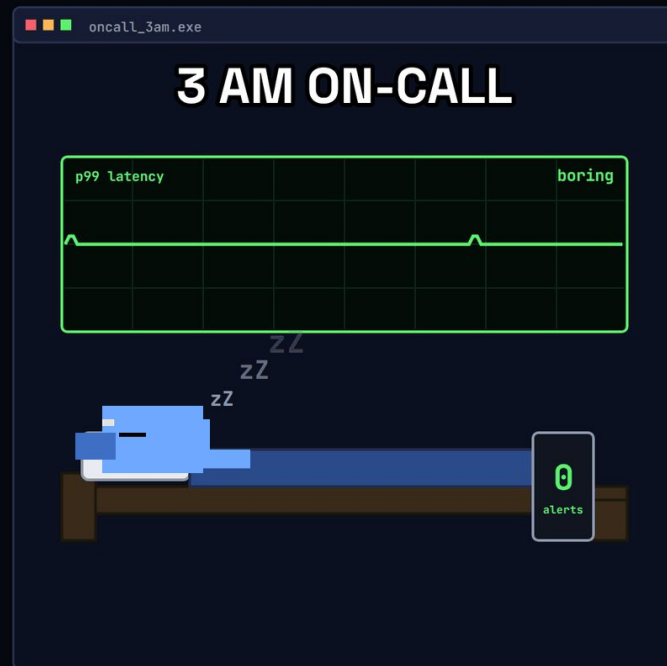
Where this actually pays off

Real-time dashboards without hammering your OLTP primary

Migrating off an expensive, closed-source pay-per-query warehouse

Keeping application response times boring and predictable

Ad-hoc analytics for the business team, without a DBA on call



Conclusion

PostgreSQL stays your source of truth - don't fight that

Pick the integration path that matches your actual latency need, not the coolest one

Start with query federation, graduate to CDC when it hurts

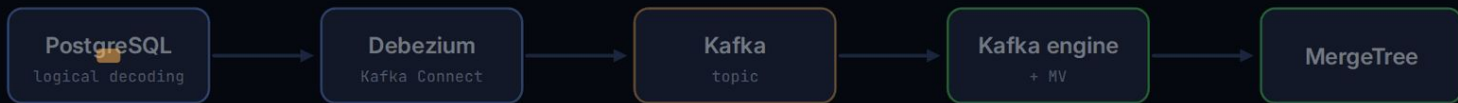
Extensibility is a benefit, not a workaround



Outlook - not covered yet

Coming soon: WalShadow

Path 4 today: Debezium + Kafka



5 hops, 3 systems to run

latency: seconds

WalShadow: ClickHouse follows the physical WAL, like an analytical standby



inside the engine:

1 track schema

2 decode in parallel

3 build native blocks

4 insert in parallel

STATUS (ClickHouse announcement, Sep 2026)

open source on GitHub today

private preview: ClickHouse Managed Postgres

needs access to the physical WAL

No replication slot, no Kafka, no JSON: the source only ships WAL, like it does for a standby

Parallel blocks may arrive out of order: every row carries its WAL position (`_lsn`), ClickHouse keeps the latest version

Outlook - not covered yet

Coming soon on Aiven



For the Docker fans

Official image, good for local dev and CI

```
docker run -d --name clickhouse-server \  
  -p 8123:8123 -p 9000:9000 \  
  --ulimit nofile=262144:262144 \  
  clickhouse/clickhouse-server  
  
docker exec -it clickhouse-server clickhouse-client
```

Bonus

Turn this into your own docker-compose stack

hub.docker.com/r/apache/kafka

official Kafka image

hub.docker.com/u/debezium

Debezium Connect images

hub.docker.com/r/clickhouse/clickhouse-server

hub.docker.com/_/postgres

Wire all four together, Your homework for tonight!





Thanks!

Questions?



Documentation and further reading

PostgreSQL

- Page layout
- Database file layout
- EXPLAIN
- Index-only scans
- Partitioning
- Parallel query
- Materialized views
- REFRESH MATERIALIZED VIEW
- COPY
- Logical decoding
- postgres_fdw
- Routine vacuuming
- High availability and replication
- Hot standby
- Logical replication
- Logical decoding output plugins

ClickHouse

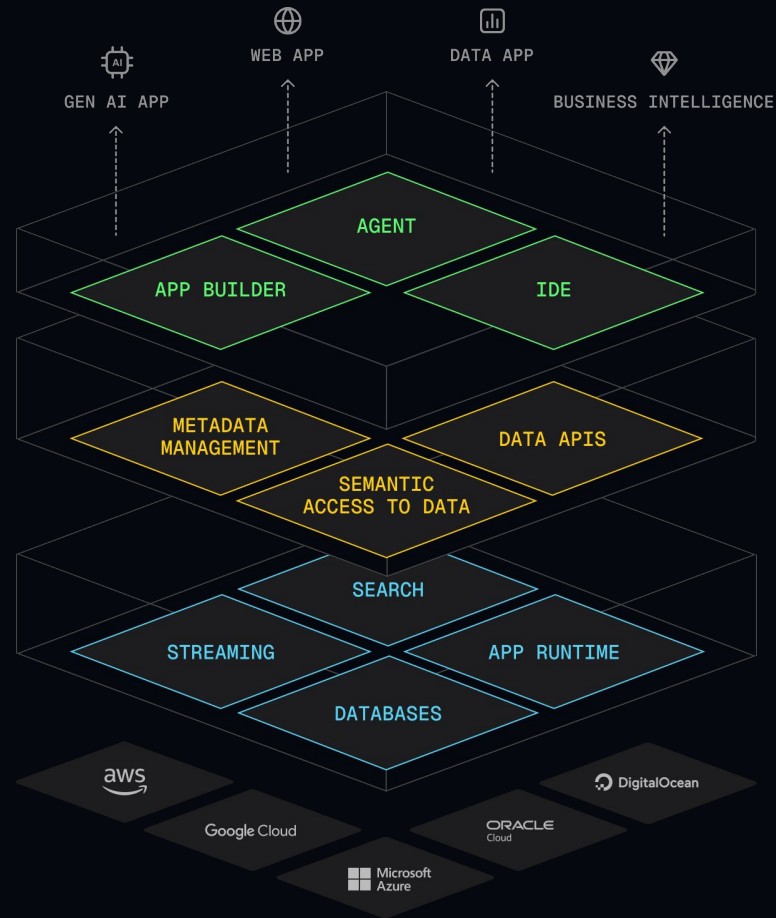
- Introduction
- Why ClickHouse is so fast
- MergeTree table engine
- Table parts
- Merges
- Sparse primary indexes
- Choosing an insert strategy
- postgresql() table function
- PostgreSQL table engine
- Kafka table engine
- Refreshable materialized views
- Materialized view best practices
- Table engines overview
- Installation
- pg_clickhouse (GitHub)
- Replicated table engines
- Distributed table engine
- WalShadow (blog)
- WalShadow (GitHub)

Aiven and friends

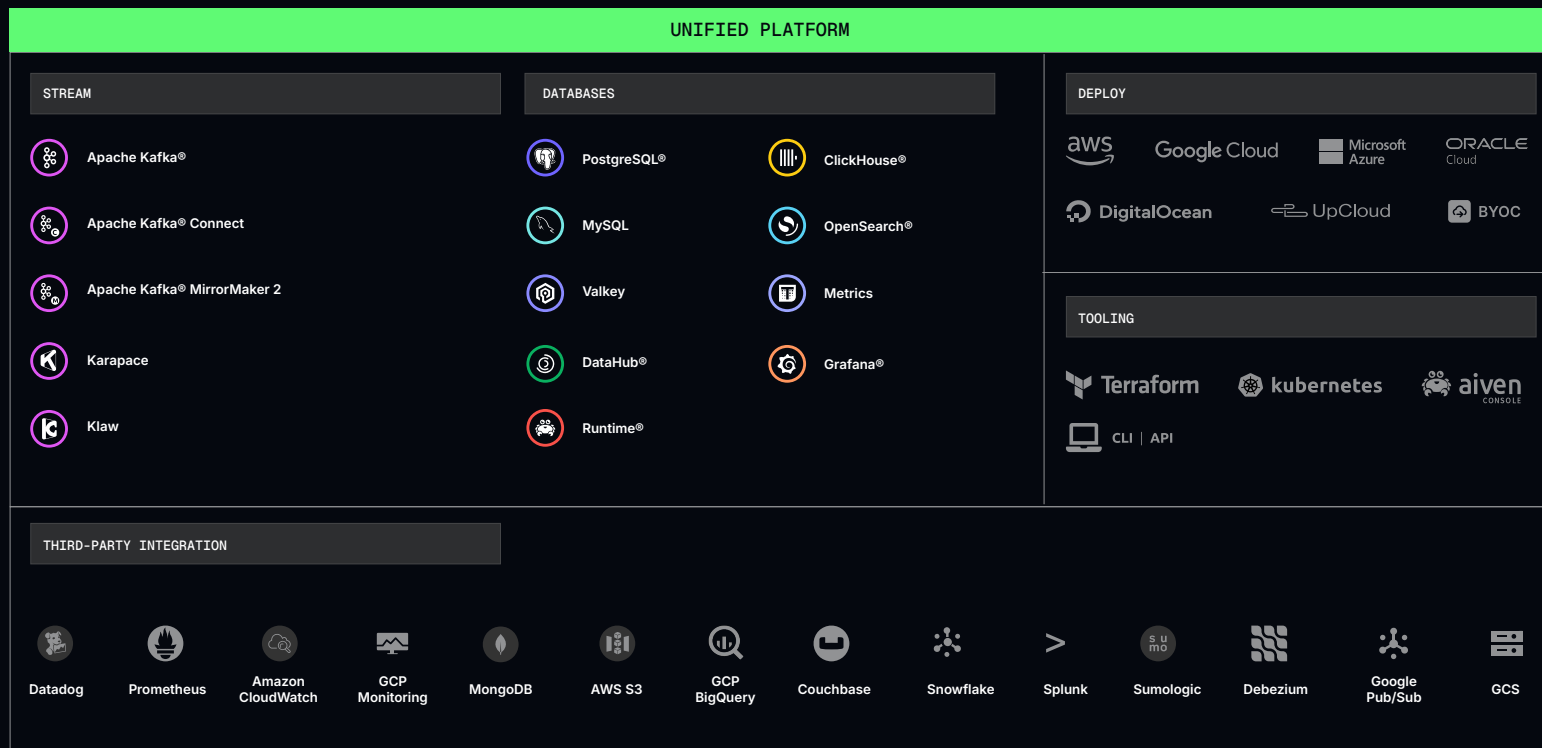
- Aiven for ClickHouse
- Integrate PostgreSQL
- Integrate Kafka
- Data integration overview
- Connect with clickhouse-client
- Aiven CLI (avn)
- aiven-client on GitHub
- Terraform overview
- Terraform: service_integration
- Aiven for PostgreSQL
- Aiven for Apache Kafka
- Debezium PostgreSQL connector
- Apache Kafka documentation
- Talk: Kafka and Debezium for DB people
- PostgreSQL read replicas
- ClickHouse service architecture
- Shards with a Distributed table

YOUR AI-READY OPEN SOURCE DATA PLATFORM

Streaming | Database Optimization |
Analytics | Search | Data Warehousing |
In-Memory Caching



AIVEN PLATFORM



TRUSTED BY OVER 1000 **CUSTOMERS** WORLDWIDE

AVAYA

idealo

Wolt

priceline®

IARE
DOU
TE

Norauto

DECATHLON

SWIFT
SOLUTIONS

energy

MIRAKL

GOV.UK

DOJO

spare

Schibsted

TOYOTA

paf

CONRAD

adeo

CLAROTY

WÄRTSILÄ