7+ million Postgres tables

Ft. DungBeetle: A hack for scaling SQL reporting on massive databases

Kailash Nadh

IndiaFOSS 2024

ZERODHA

“Abuse”

The line between a clever software hack and its outrageous abuse, is very thin.

Last year it was DNS. This time, it’s Postgres.

Very large databases

- Exist in many orgs.
- Unbounded growth. Eg: Bank transactions, eCommerce orders.
- Hot, Live, OLTP vs. Cold, OLAP, Warehouse etc.

People need reports

- 10 year old data is archived to a “warehouse” database.
- 99.9% users only need reports within ~1 year.
- 0.1% want 10 year old reports.
- Have to accommodate and provision for both.
- Ever noticed how some banks have different UIs or forms to download older statements?

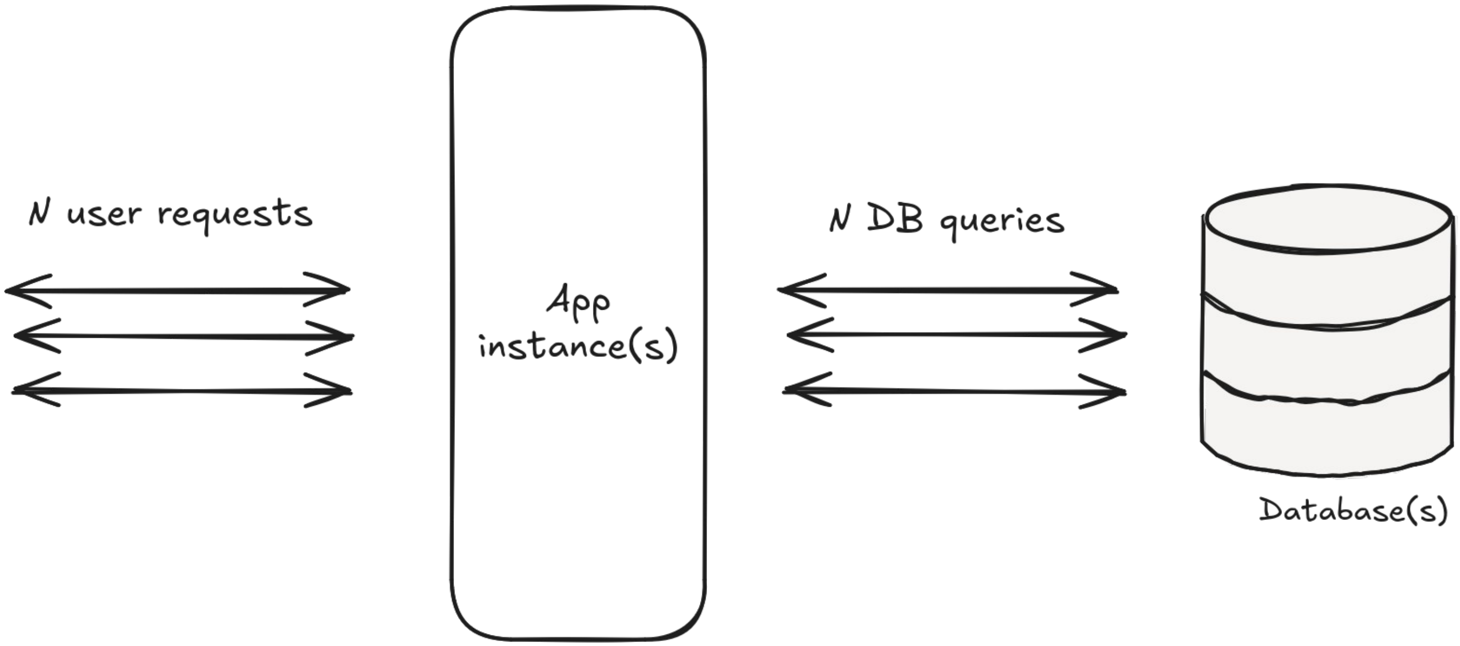
Very large DBs are slow

- At Zerodha, users download millions of reports daily.
- Some reports are as simple as:
  `SELECT * FROM tbl WHERE user_id=$1 AND date>=$2 AND date<=$3`
- Some reports are 100s of lines of SQL JOINs, aggregations etc. across multiple tables across multiple databases.
- Some of our databases (ClickHouse, Postgres) have ~100s of billions of rows.
- Some queries finish in 10ms, some 10s, some even longer.
- Thousands of reports can be requested at the same instant. User traffic is unpredictable in stock markets. Tax season / volatile stock market days = unprecedented traffic.
- It is not viable to keep provisioning read-replicas to handle queries.

Synchronous is not scalable

- ➔ User performs an action on an app and waits for results.
  - ➔ App holds that connection, sends a query to a large DB.
    - ➔ App waits for the response. User waits for the response.
      - ➔ Synchronous sequence of steps holding multiple connections.
        - ➔ Database is overloaded instantly.
          - ➔ App is overloaded next.

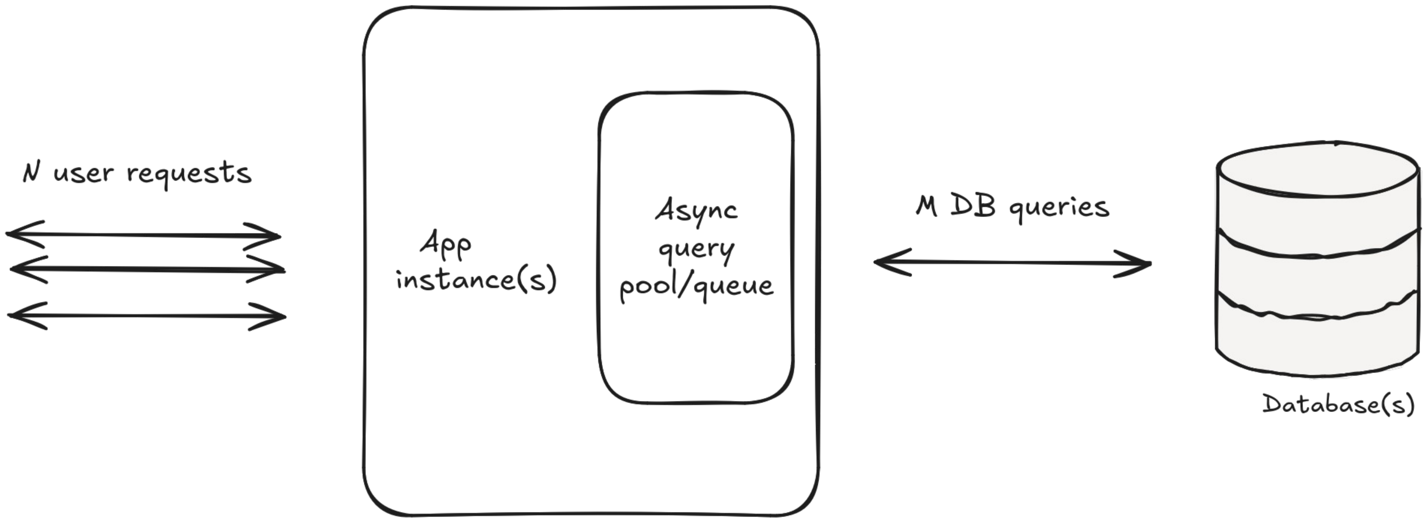
- N users can do this in the same instant.
- Cannot scale without tons of resources, difficulty, and overhead.


N user requests
App instance(s)
N DB queries
Database(s)

Async. Queue. Control.

- User requests (N) are unbounded and unpredictable but DBs have limits (M).
- Defer and pool N requests → control and send max M queries to the DB, gradually fulfilling N requests as the DBs finish them.

- This is not new. Async-queuing report-generation is an ancient technique.
- That’s why some bank UIs sometimes say:
  ~ *“Report is being generated. Please come back after a while.”*


N user requests
App instance(s)
Async query pool/queue
M DB queries
Database(s)

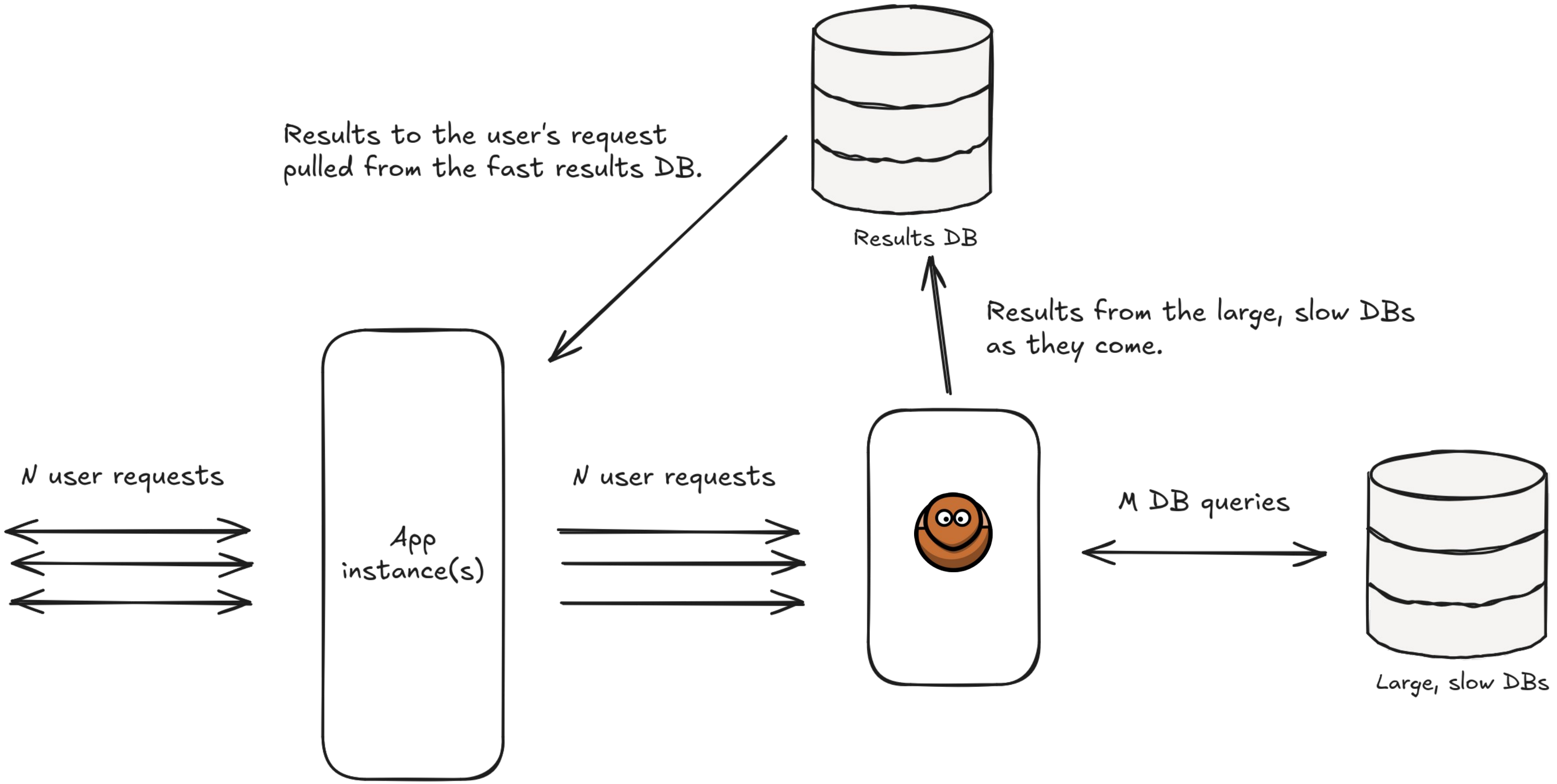
In an organisation:

- Apps can be in any language, framework, and environment.
- Data can be in any DB: Postgres, MariaDB, ClickHouse …
- Data for a single report can be spread across different kinds of DBs, eg: Postgres and ClickHouse.

Not practical to:

- Implement async pooling and queuing inside every app for every kind of report from different kinds of DBs.
- Horizontally scale or isolate app's vs. data-heavy reporting mechanism's resource usage once they are tightly coupled.

An independent middleware



DungBeetle

@zerodha/dungbeetle

- Lightweight, single binary app written in Go.
- Scalable, distributed, concurrent SQL read jobs.
- Helps separate reporting layer from apps.
- Generic, database-agnostic HTTP APIs to retrieve reports from any DB.

- Formerly, the unimaginatively named sql-jobber.
- Dung beetles are insects that can move up to ~1100x their body weight.

How it works: Tasks

“Tasks” are named queries loaded from an .sql file. The queries can generate arbitrary rows as results with any type of columns and names.

```sql
-- reports.sql

-- name: get_profit_summary
-- db: ledger-db
SELECT SUM(amount) AS total, entry_date FROM entries GROUP BY entry_date WHERE user_id = $1;

-- name: get_profit_entries_by_date
-- db: ledger-db
SELECT * FROM entries WHERE user_id = $1 AND timestamp > $2 and timestamp < $3;

-- name: get_transaction_history
-- db: tx-db
SELECT * FROM tx WHERE user_id = $2;
```

How it works: Jobs

A “job” is created in the system when a request comes in to execute a particular named query (task). get_profit_entries_by_date, the task, can be run for many users at the same time, creating a job per user which is queued and executed.

```
$ curl localhost:6060/tasks/get_profit_entries_by_date/jobs \
  -H "Content-Type: application/json" -X POST \
  --data '{"job_id": "get_profit_user1", "args": ["user1", "2015-01-01", "2015-06-30"]}'

{
 "status":"success"
  "data":{
    "job_id":"get_profit_user1",
    "task_name":"get_profit_entries_by_date",
    "queue":"queue1",
    "eta":null,"retries":0
  }
}
```

How it works: Results

- When a job is finished—when an SQL read query is executed and it returns rows—they are written to a separate results DB.
- A new table is created for each job's results in the results DB with the job ID. Column names and types in the results are automatically mapped to the results table.
- Apps poll the system for job completion status, and then simply do a SELECT * FROM get_profit_user1 from the results DB, which is instantaneous.
- Further transformations (sorting, filtering etc.) on the results can be done directly on this table.

Postgres

- Dung Beetle supports Postgres, MariaDB, and ClickHouse as source databases to read from, and results databases to write to. We picked Postgres as our results DB.
- Even on a slow day in the stock markets, people pull millions of reports, each of which creates a new table in the Postgres results DB.
- Did we know that Postgres could support millions of tables when we started out? Nope.
- Does Postgres handle it without flinching? Yep.
- A new table for every kind of report per user when they request it? Why? Why not? It works seamlessly, is highly performant, scales ridiculously well, is super cheap to run, and apps just have to do SELECT * FROM some_job_result to deliver complex reports to users without overloading large DBs.

Postgres results DB

- EC2 instance: 64 vcpu, 128 GB RAM.
- Disk (EBS volume) attached: 2 TB.
- Wiped every night.
  - Stop Postgres.
  - Detach filled volume.
  - Attach empty volume.
  - Start Postgres.
- Ready for the next day in seconds.

Stats from a random day

Tables	**~7 million**
Size on disk	~1 TB This varies wildly based on the types of reports
pg_attribute	48 GB
pg_class	9 GB
pg_index	2.5 GB
pg_statistic	566 KB
pg_constraint	128 KB
	60 GB of metadata

DungBeetle

@zerodha/dungbeetle

- Only about ~1700 lines of Go code.
- Light weight. Frugal. Single binary.
- Distributed, multi-process, multi-threaded. Run many instances (workers) that share job loads.
- Define queues for different kinds of jobs. Eg: A “heavy” job queue that has 10 workers and a “light” queue that has 2, or any such pattern.
- Group jobs: Initiate multiple jobs as a group that complete together.

```go
for i := 0; i < len(cols); i++ {
    typ = colTypes[i].DatabaseTypeName()
    switch colTypes[i].DatabaseTypeName() {
    case "INT2", "INT4", "INT8", // Postgres
        "TINYINT", "SMALLINT", "INT", "MEDIUMINT", "BIGINT": // MySQL
        typ = "BIGINT"
    case "FLOAT4", "FLOAT8", // Postgres
        "DECIMAL", "FLOAT", "DOUBLE", "NUMERIC": // MySQL
        typ = "DECIMAL"
    case "TIMESTAMP", // Postgres, MySQL
        "DATETIME": // MySQL
        typ = "TIMESTAMP"
    case "DATE": // Postgres, MySQL
        typ = "DATE"
    case "BOOLEAN": // Postgres, MySQL
        typ = "BOOLEAN"
    case "JSON", "JSONB": // Postgres
        if s.opt.DBType != dbTypePostgres {
            typ = "TEXT"
        }
    // _INT4, _INT8, _TEXT represent array types in Postgres
    case "_INT4": // Postgres
        typ = "_INT4"
    case "_INT8": // Postgres
        typ = "_INT8"
    case "_TEXT": // Postgres
        typ = "_TEXT"
    default:
        typ = "TEXT"
    }

    if nullable, ok := colTypes[i].Nullable(); ok && !nullable {
        typ += " NOT NULL"
    }

    fields[i] = fmt.Sprintf(`"%s" %s`, cols[i], typ)
}
```

Thank you

Photo courtesy of Pixabay