MongoDB vs PostgreSQL

Simone Parrella 1899329
Samuele Proietti 1946329

Project Description

The project aims to show the strengths and weaknesses of a document-based NoSQL DBMS and a relational DBMS by comparing their performance when executing queries on the same dataset. The systems we analyze are PostgreSQL and MongoDB.

Tools

DBMS	Graphic tools	Dataset used	Additional tools
PostgreSQL	pgAdmin 4	GrocerySales database	Python
MongoDB	MongoDB Compass	GrocerySales database	Python

GrocerySales Database

Link

Simulated grocery sales data from 2018-01-01 to 2018-05-09

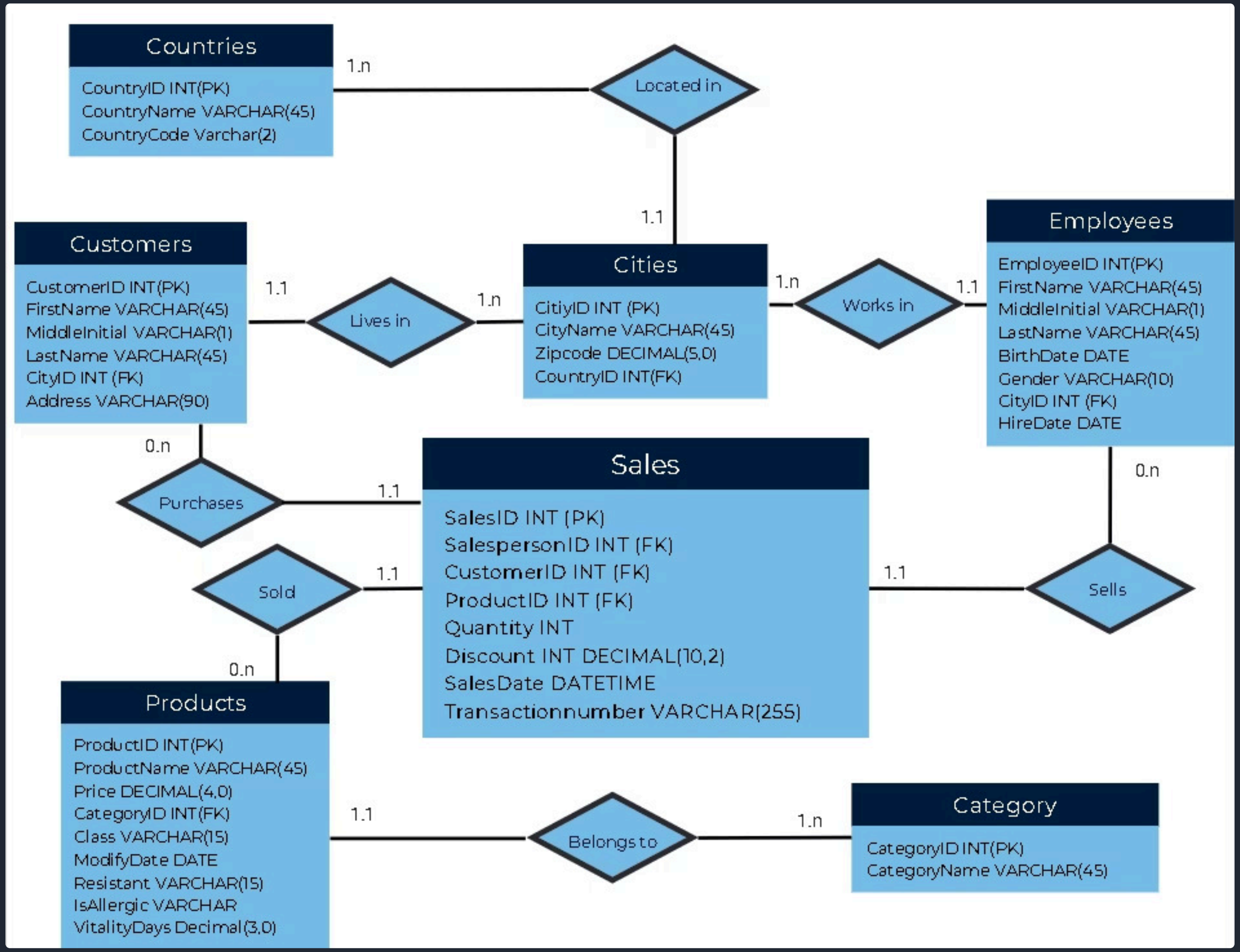
Tables

- Categories: Defines the categories of the products.
- Cities: Contains city-level geographic data.
- Countries: Stores country-related metadata.
- Customers: Contains information about the customers who make purchases.
- Employees: Stores details of employees handling sales transactions.
- Products: Stores details about the products being sold.
- Sales: Contains transactional data for each sale.

Relations:

- Purchases: A Sale is made by exactly one Customer
- Sells: A Sale is managed by exactly one Employee
- Sold: A Sale is related to exactly one Product
- Belongs to: a Product belongs to exactly one Category
- Lives in: A Customer lives in exactly one City
- Works in: An Employee works in a store located in exactly one City
- Located in: A City belongs to exactly one Country

ER Schema



Content

- 6758125 sales made in the grocery sales database each with all the details of the transaction
- 98759 Customers who make purchases
- 23 Employees who handle the transactions
- 452 Products available for sale
- 11 Categories of the products
- 96 Cities where customers, employees, and sales transactions are associated
- 206 Countries where the cities are located

PostgreSQL

Characteristics

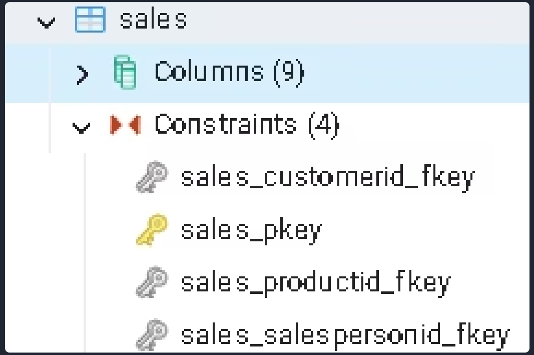
A particular advantage of using relational databases is the JOIN clause. It allows you to retrieve related data stored in multiple tables in a single command. Useful for quickly finding the needed data to complete a task and well-suited for complex queries

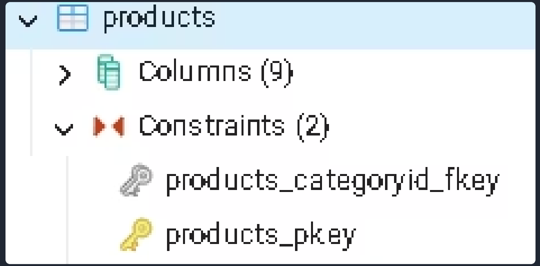
It's important that the usage is restricted to structured data. All of the data must follow the same structure and changes to the data structure would be difficult and disruptive to the whole system.

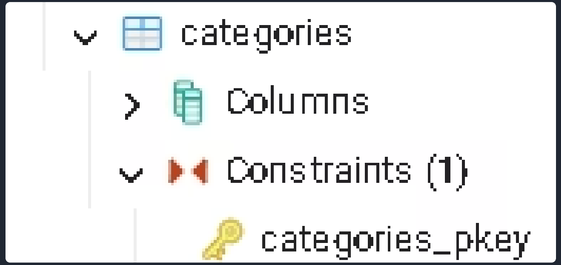
Hence we avoided applying any modifications to the original dataset for this DBMS, because it already follows a defined schema.

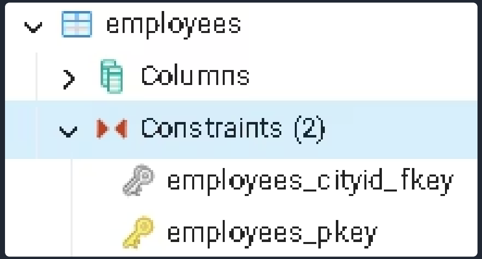
PostgreSQL

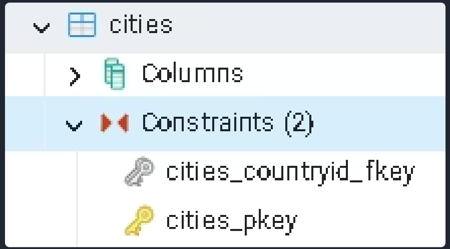
Constraints

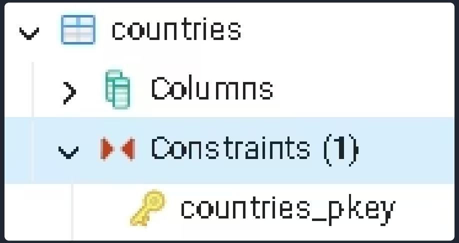

sales
Columns (9)
Constraints (4)
sales_customerid_fkey
sales_pkey
sales_productid_fkey
sales_salespersonid_fkey

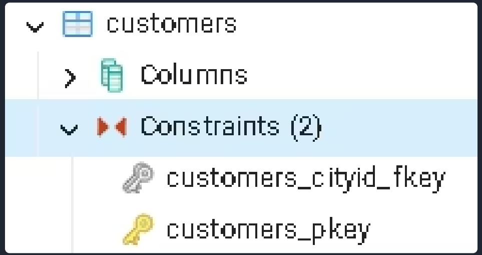

products
Columns (9)
Constraints (2)
products_categoryid_fkey
products_pkey


categories
Columns
Constraints (1)
categories_pkey


employees
Columns
Constraints (2)
employees_cityid_fkey
employees_pkey


cities
Columns
Constraints (2)
cities_countryid_fkey
cities_pkey


countries
Columns
Constraints (1)
countries_pkey


customers
Columns
Constraints (2)
customers_cityid_fkey
customers_pkey

Made with GAMMA

MongoDB Data Modeling

Denormalization

Our operations involve data coming from the collections Customers, Employees and Products that are related to the sales collection. Therefore we've denormalized the original sales collection by embedding the related data as subdocuments.

Collection	Related Data	Reason
Customer	FirstName, LastName	We want to be able to identify the customer who made the purchases
Employee	FirstName, LastName, Hiredate	We want to be able to identify the employee who handled the sale
Product	ProductName, Price	We want to be able to identify the purchased product as well as the price needed to calculate the total cost of the transaction

MongoDB Data Modeling

Denormalization

There is also partial information from collections that are related to the embedded documents, and instead of nesting another document inside the subdocuments, we've only taken the relevant fields.

Embedded document	Collection	Related Data	Reason
Customer	City	CityName	We want to know where the customer lives
Employee	City	CityName	We want to know where the employee lives
Product	Category	CategoryName	We want to know what category the product belongs to

With this approach we are able to have the most frequently used data for our queries within a single document that can be accessed with a single query without the need to perform joins. We avoided additional changes to the collection as it would have required us to update the documents, which is very complex.

MongoDB Data Modeling

Denormalization

```
{
 "_id": ObjectId,
 "Customer": {
  "CustomerID": Number,
  "FirstName": String,
  "LastName": String,
  "CityID": Number,
  "CityName": String,
  "CountryID": Number,
  "CountryName": String
 },
 "Employee": {
  "EmployeeID": Number,
  "FirstName": String,
  "LastName": String,
  "Gender": String,
  "CityID": Number,
  "CityName": String,
  "CountryID": Number,
  "CountryName": String,
  "Hiredate": Date
 },
 "Product": {
  "ProductID": Number,
  "ProductName": String,
  "CategoryID": Number,
  "CategoryName": String,
  "Price": Number
 },
 "Quantity": Number,
 "SalesDate": Date,
 "SalesID": Number,
 "TotalPrice": Number,
 "TransactionNumber": String,
 "Discount": Number
}
```

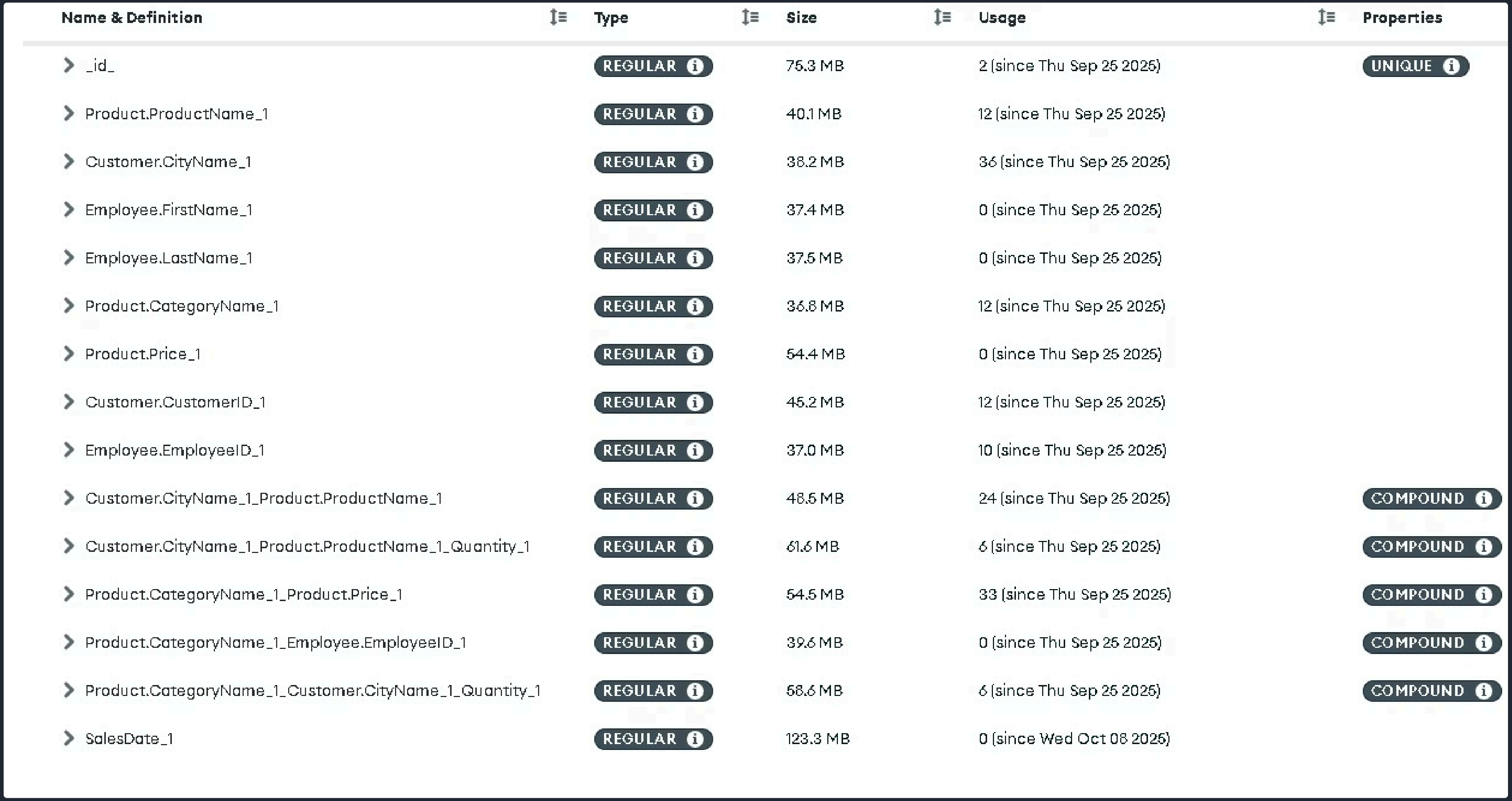
MongoDB Data Modeling

Indexes

- To further improve performance for data retrieval, we've added indexes using the fields that are the most used as search keys for range and equality queries.
- This enables the database to directly access the relevant documents, instead of scanning the entire collection.
- By minimizing the number of documents scanned, indexes lower the CPU and memory usage during query execution.
- Just like denormalization, indexes are ideal for data that is accessed frequently and is not frequently updated, because changes to indexed fields require updating the indexes as well.
- Single-field index: use when one selective predicate (equality/range) dominates the query, lowest update cost and simplest to maintain.
- Compound index: use when queries filter/sort on multiple fields, prefer covering indexes to avoid extra document fetches.

MongoDB Optimization

Indexes



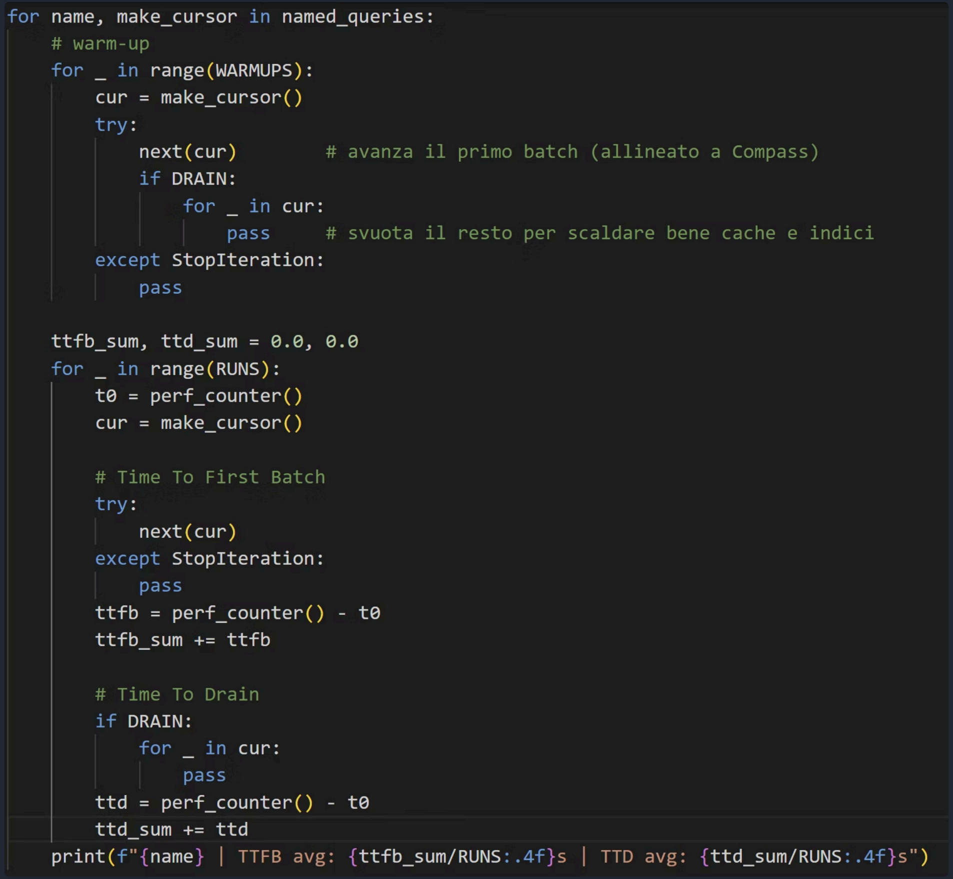
Name & Definition	Type	Size	Usage	Properties
_id_	REGULAR	75.3 MB	2 (since Thu Sep 25 2025)	UNIQUE
Product.ProductName_1	REGULAR	40.1 MB	12 (since Thu Sep 25 2025)	
Customer.CityName_1	REGULAR	38.2 MB	36 (since Thu Sep 25 2025)	
Employee.FirstName_1	REGULAR	37.4 MB	0 (since Thu Sep 25 2025)	
Employee.LastName_1	REGULAR	37.5 MB	0 (since Thu Sep 25 2025)	
Product.CategoryName_1	REGULAR	36.8 MB	12 (since Thu Sep 25 2025)	
Product.Price_1	REGULAR	54.4 MB	0 (since Thu Sep 25 2025)	
Customer.CustomerID_1	REGULAR	45.2 MB	12 (since Thu Sep 25 2025)	
Employee.EmployeeID_1	REGULAR	37.0 MB	10 (since Thu Sep 25 2025)	
Customer.CityName_1_Product.ProductName_1	REGULAR	48.5 MB	24 (since Thu Sep 25 2025)	COMPOUND
Customer.CityName_1_Product.ProductName_1_Quantity_1	REGULAR	61.6 MB	6 (since Thu Sep 25 2025)	COMPOUND
Product.CategoryName_1_Product.Price_1	REGULAR	54.5 MB	33 (since Thu Sep 25 2025)	COMPOUND
Product.CategoryName_1_Employee.EmployeeID_1	REGULAR	39.6 MB	0 (since Thu Sep 25 2025)	COMPOUND
Product.CategoryName_1_Customer.CityName_1_Quantity_1	REGULAR	58.6 MB	6 (since Thu Sep 25 2025)	COMPOUND
SalesDate_1	REGULAR	123.3 MB	0 (since Wed Oct 08 2025)	

Performance measuring

The performance was measured by considering the following metrics.

- Time to first batch: time needed for the server to send the first batch of the response to the client after executing the query.
- Time to Drain: total time needed for the client to send the request and receive the response.

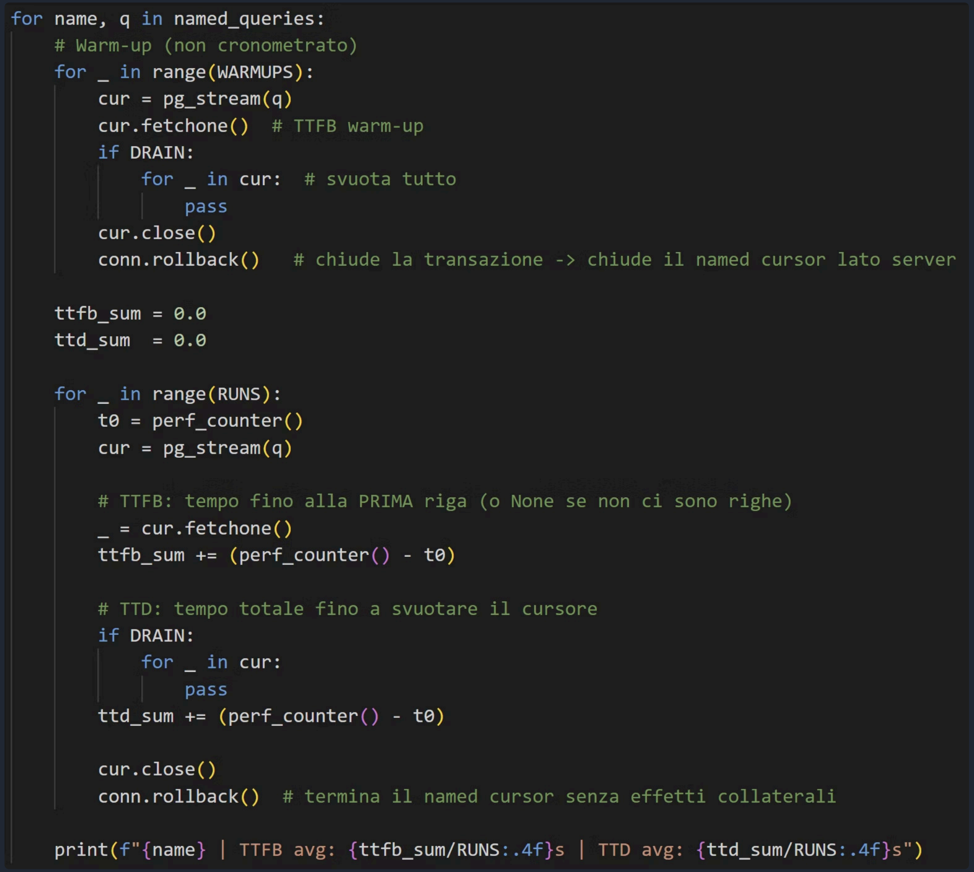
MongoDB Streaming Benchmark Algorithm



```python
for name, make_cursor in named_queries:
    # warm-up
    for _ in range(WARMUPS):
        cur = make_cursor()
        try:
            next(cur)        # avanza il primo batch (allineato a Compass)
            if DRAIN:
                for _ in cur:
                    pass     # svuota il resto per scaldare bene cache e indici
        except StopIteration:
            pass

    ttfb_sum, ttd_sum = 0.0, 0.0
    for _ in range(RUNS):
        t0 = perf_counter()
        cur = make_cursor()

        # Time To First Batch
        try:
            next(cur)
        except StopIteration:
            pass
        ttfb = perf_counter() - t0
        ttfb_sum += ttfb

        # Time To Drain
        if DRAIN:
            for _ in cur:
                pass
        ttd = perf_counter() - t0
        ttd_sum += ttd
    print(f"{name} | TTFB avg: {ttfb_sum/RUNS:.4f}s | TTD avg: {ttd_sum/RUNS:.4f}s")
```

- Create an aggregation cursor with `allowDiskUse=True` and a very large `batch_size` to reduce round-trips.
- Warm-up (not timed): open the cursor and call `next(cur)`; optionally iterate the rest to warm caches.
- Start measurement: recreate the cursor and start the timer.
- **TTFB:** time from `t0` to the first `next(cur)` (or handle `StopIteration` if empty).
- **TTD:** continue iterating until the cursor is fully drained; stop the timer at exhaustion.
- Finishing iteration closes the cursor cleanly.

PostgreSQL Streaming Benchmark Algorithm



```python
for name, q in named_queries:
    # Warm-up (non cronometrato)
    for _ in range(WARMUPS):
        cur = pg_stream(q)
        cur.fetchone()  # TTFB warm-up
        if DRAIN:
            for _ in cur:  # svuota tutto
                pass
        cur.close()
        conn.rollback()   # chiude la transazione -> chiude il named cursor lato server

    ttfb_sum = 0.0
    ttd_sum  = 0.0

    for _ in range(RUNS):
        t0 = perf_counter()
        cur = pg_stream(q)

        # TTFB: tempo fino alla PRIMA riga (o None se non ci sono righe)
        _ = cur.fetchone()
        ttfb_sum += (perf_counter() - t0)

        # TTD: tempo totale fino a svuotare il cursore
        if DRAIN:
            for _ in cur:
                pass
        ttd_sum += (perf_counter() - t0)

        cur.close()
        conn.rollback()  # termina il named cursor senza effetti collaterali

    print(f"{name} | TTFB avg: {ttfb_sum/RUNS:.4f}s | TTD avg: {ttd_sum/RUNS:.4f}s")
```

- Use a server-side named cursor with a large itersize for streaming and fewer round-trips.
- Warm-up (not timed): open the named cursor, call fetchone(); optionally iterate the rest, then close() and rollback() to cleanly end the server cursor.
- Start measurement: reopen the named cursor and start the timer.
- **TTFB:** time from t0 to the first fetchone().
- **TTD:** iterate to the end of the cursor; stop the timer when drained, then close() and rollback().
- Clean teardown each run avoids cross-run side effects.

Performance review

Queries

Specs: Intel Core Ultra 7 155H, RAM 16 GB, Intel Arc Graphics

MongoDB	TTFB	TTD	PostgreSQL	TTFB	TTD
Query 1	1.9949 s	1.9949 s	Query 1	1.0579 s	1.0583 s
Query 2	3.9090 s	3.9090 s	Query 2	2.6264 s	2.6230 s
Query 3	0.6090 s	0.6110 s	Query 3	0.8326 s	0.8335 s
Query 4	5.6767 s	5.6767 s	Query 4	1.2964 s	1.2968 s
Query 5	0.6993 s	0.6993 s	Query 5	0.8780 s	0.8784 s
Query 6	0.6768 s	0.6768 s	Query 6	0.8734 s	0.8737 s
Query 7	40.4689 s	40.4689 s	Query 7	7.7493 s	7.7496 s
Query 8	4.0426 s	4.0426 s	Query 8	1.2243 s	1.2246 s
Query 9	3.6793 s	3.6794 s	Query 9	1.2174 s	1.2178 s
Query 10	0.0035 s	0.0036 s	Query 10	0.6595 s	0.6600 s
Query 11	0.0029 s	0.0029 s	Query 11	0.6148 s	0.6152 s

Performance review

MongoDB	TTFB	TTD	PostgreSQL	TTFB	TTD
Query 12	0.1017 s	0.1018 s	Query 12	0.8036 s	0.8052 s
Query 13	12.0146 s	12.0189 s	Query 13	4.9489 s	4.9501 s
Query 14	0.5359 s	0.5372 s	Query 14	1.0271 s	1.0296 s
Query 15	0.3134 s	0.3134 s	Query 15	0.7643 s	0.7646 s
Query 16	1.9144 s	4.1563 s	Query 16	0.0018 s	1.4114 s
Query 17	0.7148 s	0.7148 s	Query 17	0.8795 s	0.8799 s
Query 18	4.1754 s	4.1960 s	Query 18	1.1953 s	1.1985 s
Query 19	0.5309 s	0.8207 s	Query 19	1.1448 s	1.1965 s
Query 20	10.7644 s	10.7644 s	Query 20	0.0005 s	0.0008 s
Query 21	8.7838 s	8.7838 s	Query 21	5.0780 s	5.0789 s
Query 22	0.7519 s	0.7520 s	Query 22	1.0155 s	1.0158 s

Query example 1

example 1

We compare two queries with very different starting selectivity.

Q6 is faster on MongoDB (0.6768 s vs 0.8734 s). A highly selective indexed filter with a small aggregation on embedded fields reads few documents and needs no joins, so the pipeline is short. PostgreSQL must join multiple tables and fetch more pages/rows even with indexes, which adds overhead and makes it a bit slower.

Q7 is faster on PostgreSQL (7.7493 s vs 40.4689 s). There is no initial filter in MongoDB, so it cannot use an index and it scans a large portion of the collection and builds big per-group DISTINCT sets, which is heavy. PostgreSQL runs the same pattern with efficient hash/sort aggregates on narrow rows, so it completes much faster.

When the first filter is highly selective and index-backed, MongoDB usually wins because it reads few documents and avoids joins. When the workload is a wide scan with heavy sorting, PostgreSQL usually wins because it's more efficient.

Query example 2

example 2

In the next case, MongoDB takes advantage of indexes for both queries. However for the first query PostgreSQL outperforms MongoDB while for the second query MongoDB is faster than PostgreSQL.

Q9 is faster on PostgreSQL (1.218 s vs 3.679 s). The category filter is low-selective, the plan uses IXSCAN on the index but examines 693,458 documents to return 10. MongoDB then builds per-employee DISTINCT sets and sorts them, which is heavy. PostgreSQL's hash/sort aggregates on narrow rows handle this pattern more efficiently.

Q12 is faster on MongoDB (0.102 s vs 0.804 s). The product filter is highly selective, IXSCAN on the index examines 15,498 documents and returns 96. With a small working set, a light group, and no joins thanks to embedded fields, the pipeline stays short.

When the index produces a small working set at the start, MongoDB tends to win; when the filter is broad and the query needs large per-group DISTINCT/sort, PostgreSQL tends to win.

	Documents returned	Documents analyzed	Examined/Returned
Query 9	10	693,458	≈ 69k×
Query 12	96	15,498	≈ 161×

Comparison

MongoDB	PostgreSQL
Flexible, document-based schema (NoSQL)	Fixed, relational schema (normalized)
Data stored in documents	Data stored in tables/rows; well-defined relationships
Related data stored in the same document (denormalization)	Well-defined relationships via structured data (normalization)
Joins generally avoided; aggregation pipelines preferred	Robust join operations and rich SQL features
Fast when most required data lives inside one document	Fast querying by retrieving related information from multiple tables
Less suitable with frequent cross-document updates	Less suitable with structural schema changes
Ideal for our analysis since it focuses on reading queries	Ideal for our dataset since it has a predefined schema

Conclusion

The results revealed distinct strengths and optimization patterns for each system, depending on the structure and complexity of the workload.

PostgreSQL achieved lower execution times in most queries involving aggregation, joins, and structured filtering.

- Its query planner and cost-based optimizer efficiently choose join strategies and exploit B-Tree and hash indexes, minimizing disk I/O access.
- PostgreSQL performs best on analytical and transactional workloads that rely on data consistency, multiple-table joins, and precise relational logic.

MongoDB outperformed PostgreSQL in document-oriented queries, especially when starting with highly selective match operations on indexed fields.

- Its denormalized data model stores all related information within a single document, eliminating costly joins and allowing subsequent aggregation stages to process a significantly reduced subset of data.
- By filtering early and exploiting in-memory aggregation and index access, MongoDB was able to minimize disk I/O and execution time, making it faster for this type of workload.

Thank you