MongoDB vs PostgreSQL

Simone Parrella 1899329
Samuele Proietti 1946329

Project Description

The project aims to show the strenght and weaknesses of a document based NoSQL DMBS and a reletional DBMS, by comparing their perfomances when executing queries on the same dataset. The System we are analyzing are PostgreSQL and MongoDB.

Tools

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

GrocerySales Database

Link

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

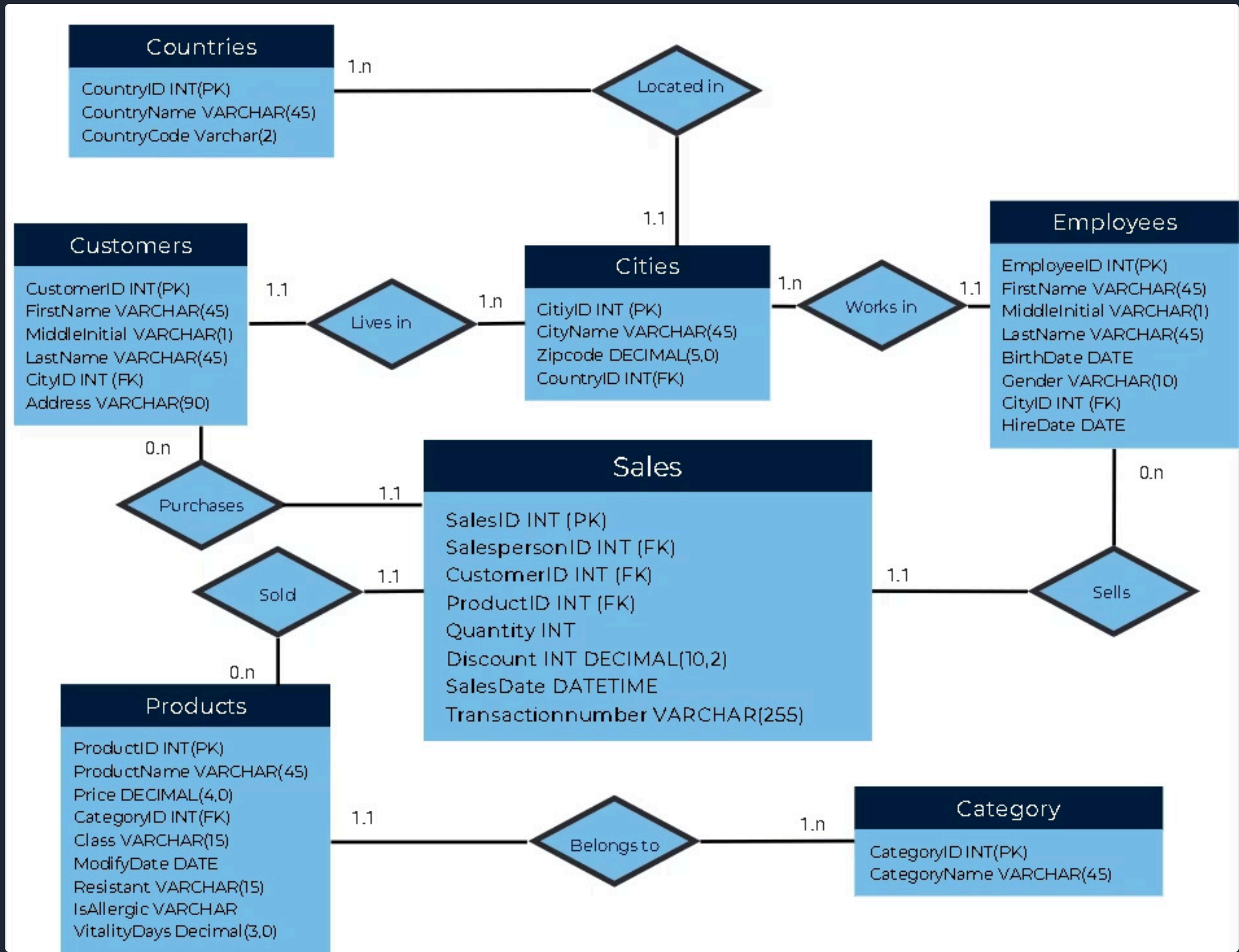
Tables

- 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



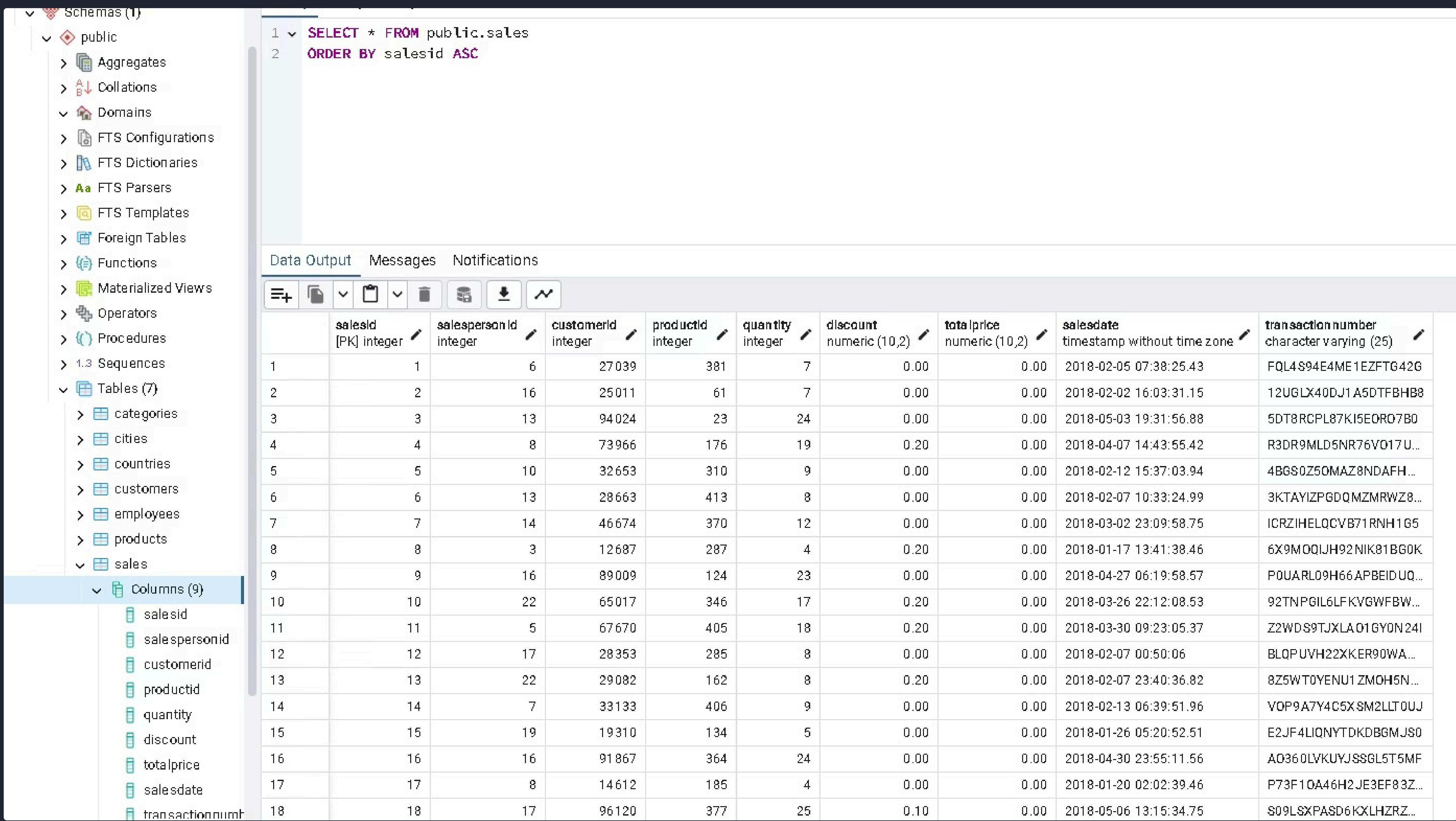
Content

- 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

Charateristics

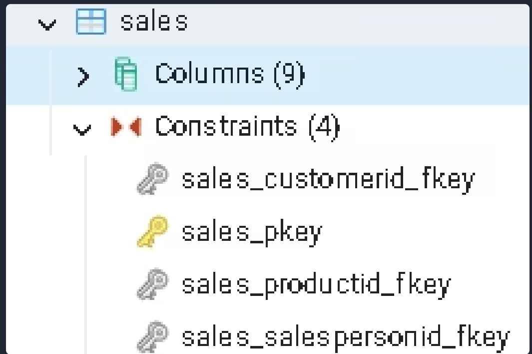
- Has a fixed schema that determines the organization of data within tables.
- Suitable for structured data, which is consistent, and relationships between tables are well-defined.
- Makes relating different types of information easy.
- A particular advantage of using relational databases is the JOIN clause. It allows to retrieve related data stored in multiple tables in a single command.
- Useful when quickly finding the data you need to complete a task and perfect for complex queries.
- Main draw back: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 modification to the original dataset for this DBMS, since it already follows a defined schema.

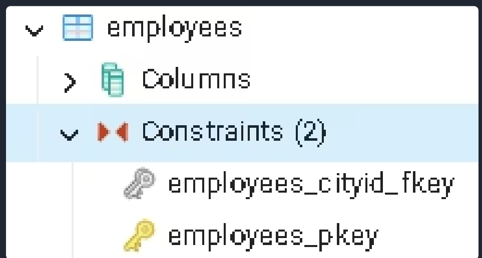


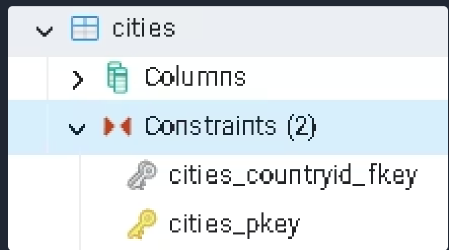
	salesid [PK] integer	salespersonid integer	customerid integer	productid integer	quantity integer	discount numeric (10,2)	totalprice numeric (10,2)	salesdate timestamp without time zone	transactionnumber character varying (25)
1	1	6	27039	381	7	0.00	0.00	2018-02-05 07:38:25.43	FQL4S94E4ME1EZFTG42G
2	2	16	25011	61	7	0.00	0.00	2018-02-02 16:03:31.15	12UGLX40DJ1A5DTFBHB8
3	3	13	94024	23	24	0.00	0.00	2018-05-03 19:31:56.88	5DT8RCPL87KI5EORO7B0
4	4	8	73966	176	19	0.20	0.00	2018-04-07 14:43:55.42	R3DR9MLD5NR76VO17U...
5	5	10	32653	310	9	0.00	0.00	2018-02-12 15:37:03.94	4BGS0Z5OMAZ8NDAFH...
6	6	13	28663	413	8	0.00	0.00	2018-02-07 10:33:24.99	3KTAYIZPGDQMZMRWZ8...
7	7	14	46674	370	12	0.00	0.00	2018-03-02 23:09:58.75	ICRZIHELQCVB71RNH1G5
8	8	3	12687	287	4	0.20	0.00	2018-01-17 13:41:38.46	6X9MOQIJH92NIK81BG0K
9	9	16	89009	124	23	0.00	0.00	2018-04-27 06:19:58.57	P0UARL09H66APBEIDUQ...
10	10	22	65017	346	17	0.20	0.00	2018-03-26 22:12:08.53	92TNPGIL6LFKVGWFBW...
11	11	5	67670	405	18	0.20	0.00	2018-03-30 09:23:05.37	Z2WDS9TJXLA01GY0N24I
12	12	17	28353	285	8	0.00	0.00	2018-02-07 00:50:06	BLQPUVH22XKER90WA...
13	13	22	29082	162	8	0.20	0.00	2018-02-07 23:40:36.82	8Z5WT0YENU1ZMOH5N...
14	14	7	33133	406	9	0.00	0.00	2018-02-13 06:39:51.96	VOP9A7Y4C5XSM2LLT0UJ
15	15	19	19310	134	5	0.00	0.00	2018-01-26 05:20:52.51	E2JF4LIQNYTDKDBGMJS0
16	16	16	91867	364	24	0.00	0.00	2018-04-30 23:55:11.56	AO360LVKUYJSSGL5T5MF
17	17	8	14612	185	4	0.00	0.00	2018-01-20 02:02:39.46	P73F10A46H2JE3EF83Z...
18	18	17	96120	377	25	0.10	0.00	2018-05-06 13:15:34.75	S09LSXPASD6KXLHZRZ...

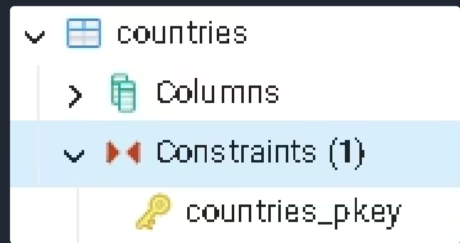
PostgreSQL

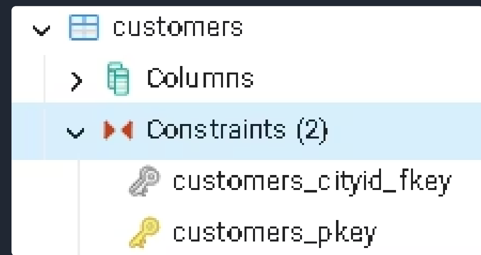
Constraints











MongoDB Optimazation

Denormalazation

- MongoDB offers flexible schema design.
- While it's possible to reference related data in like in relational Databases ,it also offers the option for denormalization.
- Related data is stored in the same document instead of separate collections .
- Improves read performance by retrieving all information with single queries.
- Avoids the need for joins operations.
- Main drawback: write operations requiring updates across multiple documents which can be very complex .
- Best used when accessing frequently read data with infrequent updates.
- Suitable for our analysis since it focuses on reading queries.

MongoDB Optimazation

Denormalization


```
_id: ObjectId('682df36872b7e36c3fd54e16')
SalesID : 1
SalesPersonID : 6
CustomerID : 27039
ProductID : 381
Quantity : 7
Discount : 0
TotalPrice : 0
SalesDate : "2018-02-05 07:38:25.430"
TransactionNumber : "FQL4S94E4ME1EZFTG42G"
```

```
_id: ObjectId('682df36872b7e36c3fd54e16')
Customer : Object
    CustomerID : 27039
    FirstName : "Susan"
    LastName : "Green"
    CityID : 54
    CityName : "Albuquerque"
    CountryID : 32
    CountryName : "United States"
Discount : 0
Employee : Object
    EmployeeID : 6
    FirstName : "Holly"
    LastName : "Collins"
    Gender : "M"
    CityID : 65
    CityName : "Baltimore"
    CountryID : 32
    CountryName : "United States"
    Hiredate : 1970-01-01T00:00:00.000+00:00
Product : Object
    ProductID : 381
    ProductName : "Vaccum Bag 10x13"
    Price : 44.2337
    CategoryID : 1
    CategoryName : "Confections"
Quantity : 7
SalesDate : "2018-02-05 07:38:25.430"
SalesID : 1
TotalPrice : 0
TransactionNumber : "FQL4S94E4ME1EZFTG42G"
```

MongoDB Optimazation

Indexes

- To further improve perfomance 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 acessed frequently and is not subjected to updates, since if the changes involve the fields of the indexes, they will need to be updated as well.

MongoDB Optimazation

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)	

Perfomance measuring

The performance was done 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 recieve 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 (opzionale)
        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:** restart 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

```
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.

Benchmark Results: TTFB & TTD

Queries

```
PS C:\Users\samue\OneDrive\Documenti\Università Magistrale\Data Management\DM-project> python test_avg_NOSQL.py
Q1 | TTFB avg: 1.9949s | TTD avg: 1.9949s
Q2 | TTFB avg: 3.9090s | TTD avg: 3.9090s
Q3 | TTFB avg: 0.6090s | TTD avg: 0.6110s
Q4 | TTFB avg: 5.6767s | TTD avg: 5.6767s
Q5 | TTFB avg: 0.6993s | TTD avg: 0.6993s
Q6 | TTFB avg: 0.6768s | TTD avg: 0.6768s
Q7 | TTFB avg: 40.4689s | TTD avg: 40.4689s
Q8 | TTFB avg: 4.0426s | TTD avg: 4.0426s
Q9 | TTFB avg: 3.6793s | TTD avg: 3.6794s
Q10 | TTFB avg: 0.0035s | TTD avg: 0.0036s
Q11 | TTFB avg: 0.0029s | TTD avg: 0.0029s
Q12 | TTFB avg: 0.1017s | TTD avg: 0.1018s
Q13 | TTFB avg: 12.0146s | TTD avg: 12.0189s
Q14 | TTFB avg: 0.5359s | TTD avg: 0.5372s
Q15 | TTFB avg: 0.3134s | TTD avg: 0.3134s
Q16 | TTFB avg: 1.9144s | TTD avg: 4.1563s
Q17 | TTFB avg: 0.7148s | TTD avg: 0.7148s
Q18 | TTFB avg: 4.1754s | TTD avg: 4.1960s
Q19 | TTFB avg: 0.5309s | TTD avg: 0.8207s
Q20 | TTFB avg: 10.7644s | TTD avg: 10.7644s
Q21 | TTFB avg: 8.7838s | TTD avg: 8.7838s
Q22 | TTFB avg: 0.7519s | TTD avg: 0.7520s
```

f

```
PS C:\Users\samue\OneDrive\Documenti\Università Magistrale\Data Management\DM-project> python test_avg_SQL.py
Q1 | TTFB avg: 1.0579s | TTD avg: 1.0583s
Q2 | TTFB avg: 2.6224s | TTD avg: 2.6230s
Q3 | TTFB avg: 0.8326s | TTD avg: 0.8335s
Q4 | TTFB avg: 1.2964s | TTD avg: 1.2968s
Q5 | TTFB avg: 0.8780s | TTD avg: 0.8784s
Q6 | TTFB avg: 0.8734s | TTD avg: 0.8737s
Q7 | TTFB avg: 7.7493s | TTD avg: 7.7496s
Q8 | TTFB avg: 1.2243s | TTD avg: 1.2246s
Q9 | TTFB avg: 1.2174s | TTD avg: 1.2178s
Q10 | TTFB avg: 0.6595s | TTD avg: 0.6600s
Q11 | TTFB avg: 0.6148s | TTD avg: 0.6152s
Q12 | TTFB avg: 0.8036s | TTD avg: 0.8052s
Q13 | TTFB avg: 4.9489s | TTD avg: 4.9501s
Q14 | TTFB avg: 1.0271s | TTD avg: 1.0296s
Q15 | TTFB avg: 0.7643s | TTD avg: 0.7646s
Q16 | TTFB avg: 0.0018s | TTD avg: 1.4114s
Q17 | TTFB avg: 0.8795s | TTD avg: 0.8799s
Q18 | TTFB avg: 1.1953s | TTD avg: 1.1985s
Q19 | TTFB avg: 1.1448s | TTD avg: 1.1965s
Q20 | TTFB avg: 0.0005s | TTD avg: 0.0008s
Q21 | TTFB avg: 5.0780s | TTD avg: 5.0789s
Q22 | TTFB avg: 1.0155s | TTD avg: 1.0158s
```

Performance review

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

MongoDB	TTFB	TTD	Postgres	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

Perfomance review

MongoDB	TTFB	TTD	Postgres	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

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 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.

Comparison

MongoDB	PostgreSQL
Flexible schema (NoSQL)	Fixed schema (Relational)
Data organized in documents	Data organized in tables/rows
Related data stored in the same document (denormalization)	Well-defined relationships via structured data (normalization)
Joins are generally avoided	Supports robust JOIN operations
Fast querying by retrieving related information from single documents	Fast querying by retrieving related information from multiple tables
Less optimal with frequent updates to collections	Less optimal with structural schema changes
Ideal for our analysis since it focuses on reading queries	Ideal for our dataset since it has a pre-defined schema

Thank you