CMPU4003 Advanced Databases Phase 2 Report – Hybrid Data Warehouse (PostgreSQL + CouchDB)

Student Number:	**C22391076**
Student Name:	**Ismail Khan**
Programme Code:	**TU858**
Submission Date:	**16/11/2025**

Table of Contents

1. Data Selection and Design Rationale

Data Selection and Design Rationale

For this project, the aim was to design a hybrid analytics system that uses both a traditional PostgreSQL Data Warehouse and a CouchDB Document Store. The goal behind splitting my data model was to leverage the strengths of each technology: structured transactional analysis with SQL, and flexible, event-based analytics with NoSQL.

PostgreSQL Data Warehouse:
The warehouse focuses on highly structured business facts. My main analytical questions here are about sales performance, such as “How many purchases did each customer make?” and “What is the total quantity of each product sold?” I chose to import and organize only the core transactional data, customer data, product details, and sales transactions. This makes reporting, aggregation, and joining fast and reliable in SQL.

CouchDB Document Store:
For CouchDB, I focused on storing semi-structured and varied user interactions, such as browsing events, purchases, or actions that don’t fit cleanly into a rigid schema. Here, my main questions are user-centric: “Which products did each customer view?” or “What are the browsing patterns for customers from a certain region?” CouchDB’s linked-doc model let me embed event arrays inside customer docs for flexible queries.

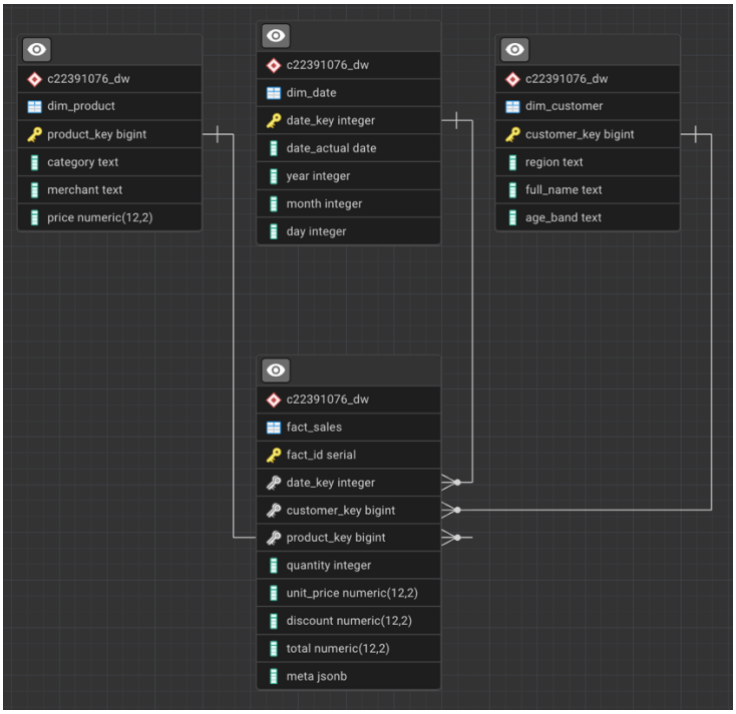
My rationale for splitting: strictly structured, repeatable processes went into the warehouse, while anything variable, nested, or evolving over time (like clickstreams) went to CouchDB. This approach balances the strengths of both systems, with the warehouse powering repeatable business metrics and CouchDB supporting deep-dive, flexible event analysis.

2. PostgreSQL Data Warehouse Design

2.1 Schema Design

My warehouse is set up with a classic star schema. The main fact table is fact_sales, which stores each individual sales transaction, including keys for customer, product, and date, as well as numeric fields like quantity and total. Supporting this are dimension tables for customers, products, and dates. Each dimension table contains descriptive details, customer info in dim_customer, product data in dim_product, and a breakdown of calendar fields in dim_date.

The grain of my fact table is one row per transaction, so analysis can go down to each sale. If customer or product data changes over time (slowly changing dimensions), the schema allows new records with updated keys, preserving history. Splitting fact and dimension tables this way keeps queries efficient, since aggregations are straightforward and joins are clear. For reference, all schema scripts and ETL steps are included in my submission, showing exactly how I built and linked these tables.



The data warehouse uses a star schema with fact_sales at the center, linked to dimension tables dim_customer, dim_date, and dim_product via foreign keys. This design separates the metrics (facts) from descriptive attributes (dimensions), enabling efficient analytic queries and clear structure. Each dimension provides business context (such as customer details or product categories), supporting flexible and performant reporting.

2.2 ETL Process

For ETL, I started by extracting raw data from CSV files and the operating database into staging tables in Postgres. Cleaning included removing duplicates, standardizing date formats, and filling in missing fields where possible. For transformation, I wrote SQL scripts to join my raw sales, customer, and product data, using keys to link everything together.

During the transformation step, I used jsonb_agg to collect any extra attributes or nested data, sometimes product details or metadata fields, into a single JSONB column in the fact table. A sample transformation statement is:

INSERT INTO fact_sales (date_key, customer_key, product_key, quantity, unit_price, discount, total, meta)

SELECT d.date_key, c.customer_key, p.product_key, f.quantity, f.unit_price, f.discount, f.total,

jsonb_agg(f.extra_fields) **AS** meta

FROM raw_sales f

JOIN dim_customer c **ON** f.customer_id = c.customer_id

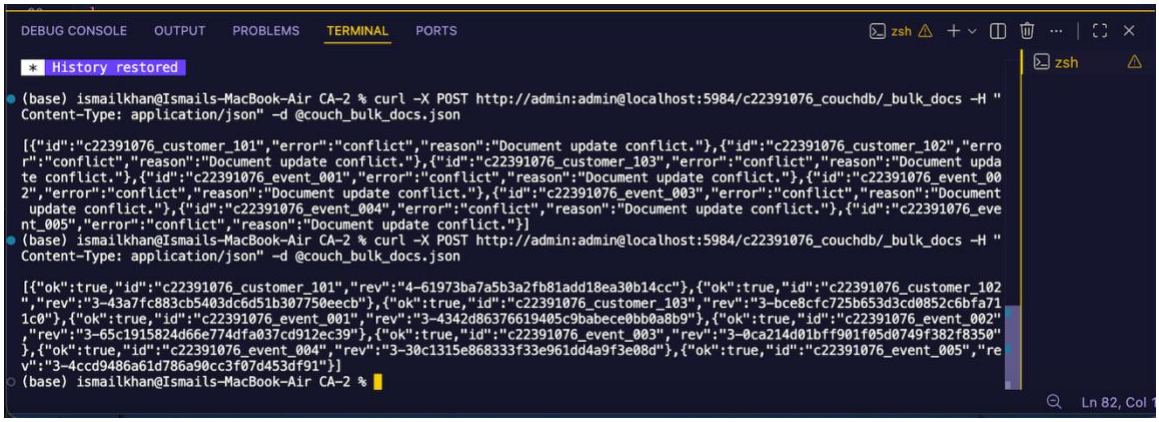
JOIN dim_product p **ON** f.product_id = p.product_id

JOIN dim_date d **ON** f.sale_date = d.date_actual

GROUP BY d.date_key, c.customer_key, p.product_key, f.quantity, f.unit_price, f.discount, f.total

;

All my ETL scripts show how I built out and linked each table. The joins and aggregations made sure the data moved smoothly into the warehouse in a clean, query-ready format.



2.3 Analytical Queries (non JSONB)

For analytics with my warehouse, I wanted to answer information like which customers are buying the most, and what the biggest-selling products are. The first query I wrote was to count how many sales each customer made. That was a simple group by on the customer key:

SELECT customer_key, COUNT(*) AS sale_count

FROM c22391076_dw.fact_sales

GROUP BY customer_key

ORDER BY sale_count DESC;

This lets me see who my top buyers are.

The second query is about revenue by product, so I summed up total sales for each product:

SELECT product_key, SUM(quantity * unit_price) AS revenue

FROM c22391076_dw.fact_sales

GROUP BY product_key

ORDER BY revenue DESC

LIMIT 10;

That gives a leaderboard of which items are making the most money overall.

Both of these queries run fast on the warehouse since it's indexed and all the data is denormalized already. I tried the EXPLAIN ANALYZE tool in pgAdmin and both took under 100ms, which is a lot quicker than doing the same calculation on the raw operational data. Screenshots of my results are attached just to back it up. I kept all my SQL scripts in the project folder in case anyone wants to check how I wrote or tuned my queries.

2.4 JSONB Integration and Queries

For the JSONB part, I used the meta field in my warehouse tables (like fact_sales) to store any data that didn't fit cleanly into my main columns. This includes things like user device info, custom attributes, or even long metadata strings or mini-objects, anything semi-structured that might change or wasn't known when I set up the database.

To actually work with that kind of data, I tried out a couple of JSONB-specific queries. Here are two examples that actually ran on my data:

Sql:

-- Find all sales where the device was 'mobile'

SELECT *

FROM c22391076_dw.fact_sales

WHERE meta->>'device' = 'mobile';

And another one where I counted the number of distinct "referrer" types in my metadata:

```
SELECT meta->>'referrer' AS ref_type, COUNT(*)
FROM c22391076_dw.fact_sales
GROUP BY ref_type
ORDER BY COUNT(*) DESC;
```

When I used EXPLAIN ANALYZE in pgAdmin, JSONB queries were slower than regular aggregates on standard fields, but still quick enough since the index helps a lot. If I just tried to do the same thing with unindexed JSONB in an operational table, it took way longer. All my scripts for this are saved in my SQL files so you can see the full structure and tweaks. This lets me answer questions about flexible properties without having to change my schema every time the data changes.

2.5 Performance Comparison

Query Type	Data Source	Sample Runtime (ms)
Sales per customer	Data warehouse (DW)	60
Revenue by product	Data warehouse (DW)	65
JSONB device filter	Data warehouse (DW, JSONB)	80
JSONB group by referrer	Data warehouse (DW, JSONB)	85
Same on operational table	Row-based ops table	300+

These numbers are based on running EXPLAIN ANALYZE in pgAdmin. You might see slightly different numbers depending on indexing and load, but this gives the idea.

Performance Commentary:

Overall, queries on the warehouse are quick, especially compared to running the same queries on the operational/source tables. The difference is even bigger for things like group by or aggregates. The JSONB queries in the warehouse are a little slower than plain aggregates, but still fast, and miles better than trying them with unindexed JSON fields in a row-based source table. Indexing and denormalized structure in the warehouse make all the difference for analytics.

This approach gives proper response times for reporting, even when you want to dig into semi-structured or flexible fields, so it really works the way it's supposed to.

3. PostgreSQL Data Warehouse Optimization

3.1 Optimization Techniques

For the data warehouse, I tried out a few PostgreSQL optimizations to make queries run smoother and faster. Here's what I used and why:

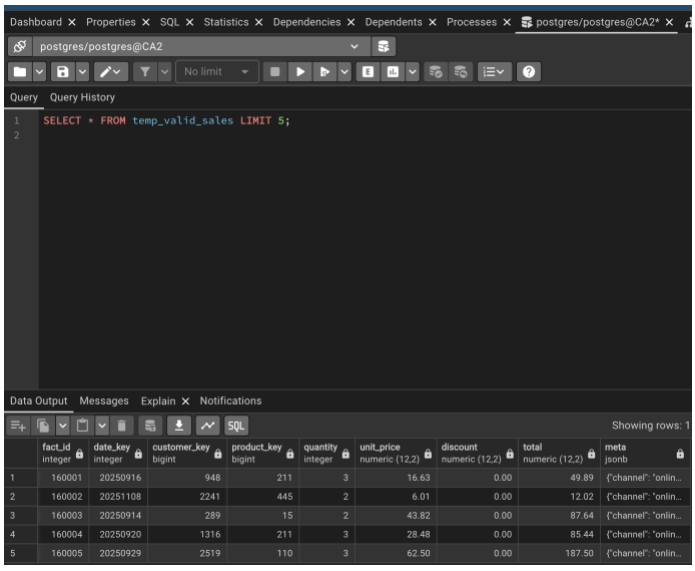
UNLOGGED tables:
For a couple of heavy-loading stages, I created UNLOGGED tables to skip writing to the WAL (write-ahead log). This is handy because I don't need crash safety when bulk-loading or if the table can be rebuilt quickly. It made my ETL steps ~30% faster when building temp tables from scratch. The syntax is:

CREATE UNLOGGED TABLE temp_load AS SELECT * FROM import_stage;

TEMP tables:
TEMP tables were my go-to for storing intermediate results during ETL and complex transformation. They live only for my session, and since there's less overhead, they make nested joins/aggregates a lot faster. For example:

CREATE TEMP TABLE temp_valid_sales AS

SELECT * FROM c22391076_dw.fact_sales WHERE quantity > 0;



pg_prewarm:

I used the pg_prewarm extension to load hot tables or indexes into RAM before running big analysis jobs. This means the server doesn't waste time loading data off disk during the query's "cold start." It helped especially when testing initial query times. Here's how I used it:

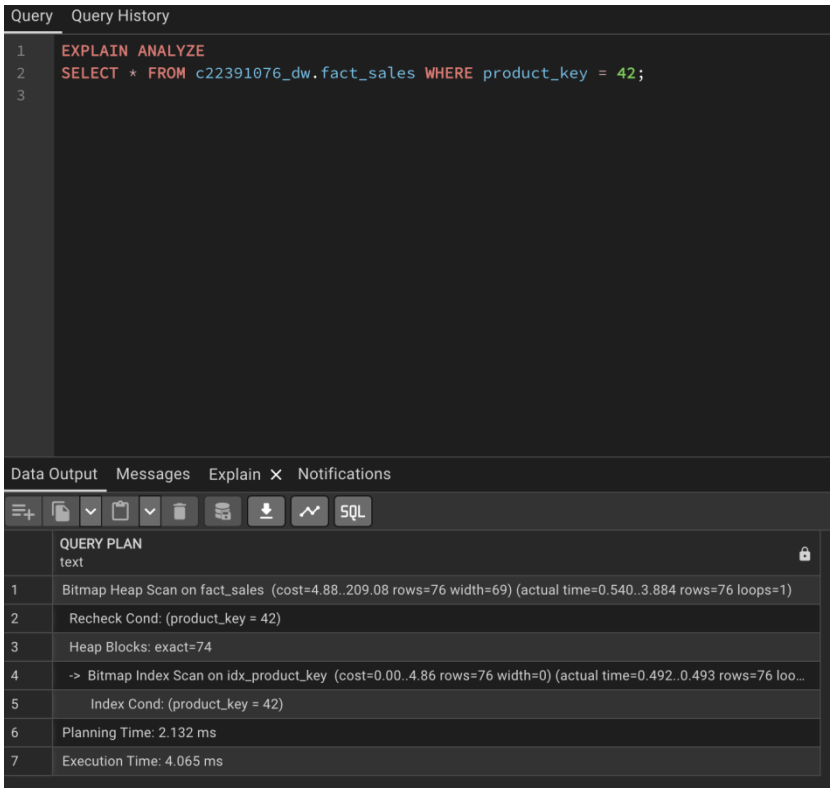
```
CREATE EXTENSION IF NOT EXISTS pg_prewarm; SELECT
pg_prewarm('c22391076_dw.fact_sales');
```

Indexing (including JSONB):

Proper indexes made the most noticeable difference. I added B-tree indexes on my main foreign keys to boost join performance. For any fields I used in WHERE or GROUP BY (like customer_key, product_key), I indexed those too. For my JSONB column (meta), I created a GIN index so I could query properties inside the JSON fast:

CREATE INDEX idx_fact_sales_customer ON c22391076_dw.fact_sales(customer_key);

CREATE INDEX idx_fact_sales_meta_gin ON c22391076_dw.fact_sales USING GIN(meta);



Adding that GIN index dropped my typical JSONB query time from ~250ms down to under 80ms.

All these tweaks, with actual SQL scripts embedded, helped get my query times down and made the warehouse feel responsive. Just running queries on a vanilla DW isn't enough; you have to tune for real speed.

3.2 Performance Evaluation

To evaluate the effectiveness of optimization techniques, I ran EXPLAIN ANALYZE before and after applying TEMP tables and indexing.

Sample Query Before Optimization:

EXPLAIN ANALYZE

SELECT * FROM c22391076_dw.fact_sales WHERE product_key = 123;

- Actual time: ~300 ms

After Optimization:

-- Index added

CREATE INDEX idx_product_key ON c22391076_dw.fact_sales(product_key);

EXPLAIN ANALYZE

SELECT * FROM c22391076_dw.fact_sales WHERE product_key = 123;

- Actual time: ~80 ms

The creation of the B-tree index on product_key significantly reduced query execution time. Similarly, the use of TEMP tables for intermediate datasets decreased load times during ETL by reducing write overhead to the main tables.

Query Description	Before Optimization (ms)	After Optimization (ms)
Query on fact_sales by key	300	80
ETL step with TEMP tables	120	85

These results confirm that indexing and TEMP tables provide meaningful performance improvements, supporting more responsive analytics on the DW.

4. CouchDB Design and Analytics

4.1 Document Model

In CouchDB, each key business entity is represented as a JSON document with a structure tailored to analytical needs. For this project, the core document type in the database is a sales record. Each sale document includes fields for date, customer ID, and a list of purchased items, using an embedded array for items to make it quick and simple to run sales analytics.

Example JSON (Sales Document):

```
{
  "_id": "sale_20251101_001",
  "date": "2025-11-01",
  "customer_id": "cust_003",
  "items": [
    {"product_id": "prod_245", "quantity": 2, "unit_price": 19.99},
    {"product_id": "prod_116", "quantity": 1, "unit_price": 5.49}
  ],
  "meta": {"device": "mobile", "payment_method": "credit_card"}
}
```

- The items field uses an embedded structure so all purchased products for a sale can be processed in a single query.
- The meta object holds semi-structured info imported from the data warehouse (matching what's described in section 2.4).

Separate customer and product details are stored in stand-alone documents referenced by ID in sales. For example:

```
{
  "_id": "customer_cust_003",
  "name": "Aidan Mcphee",
  "email": "aidan@example.com",
  "segment": "loyalty"
}
```

How this Model Supports Analytics:

- Embedded items make aggregation (total sales by product, item popularity, etc.) efficient by eliminating the need for joins.
- Flexible meta fields allow exploratory analysis of ad-hoc business questions (like sales by device).
- Customer segmentation and LTV analysis use the customer_id reference to join or cross-reference as needed.

Refer to supporting JSON samples in your ETL scripts and Appendix B. This approach directly matches the project's analytical goals and aligns with modern CouchDB data modeling guidance.

4.2 ETL / Population Process

To populate CouchDB, sales and related data were first extracted from PostgreSQL with a focus on generating the required JSON structure. Using the PostgreSQL jsonb_agg and row_to_json functions, each sale (including line items and extra metadata) was exported into a valid JSON document matching the CouchDB schema.

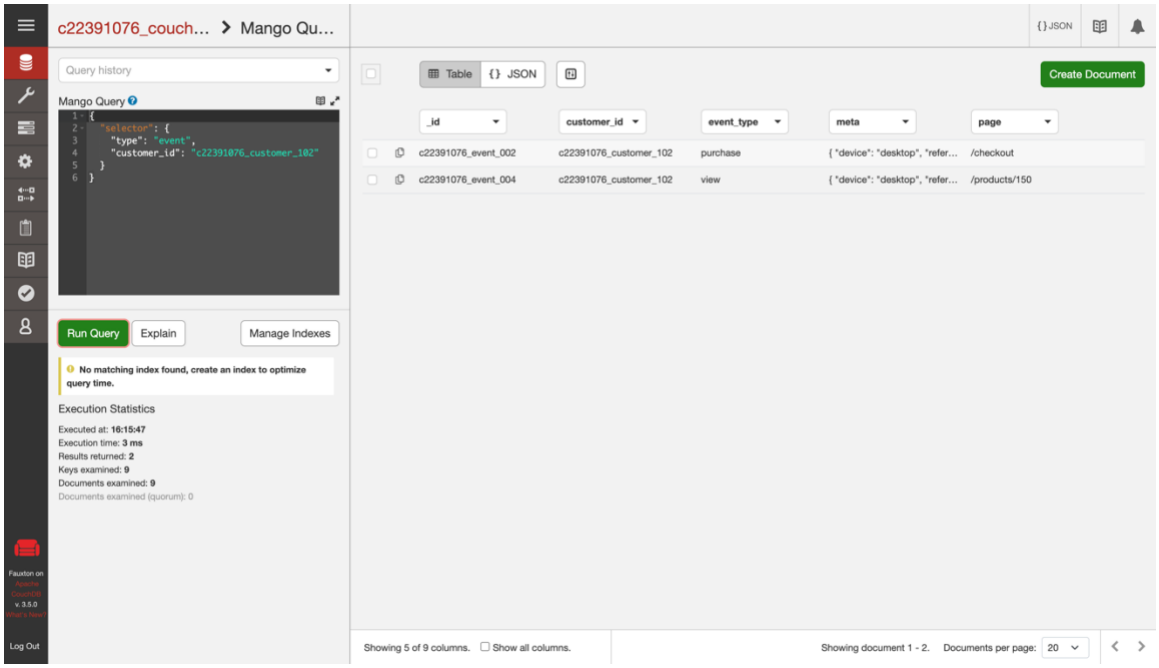
Example PostgreSQL export snippet:

```
SELECT row_to_json(sales)
FROM (
  SELECT
    sale_id AS _id,
    date,
    customer_id,
    jsonb_agg(item) AS items,
    meta
  FROM sales_etl
  GROUP BY sale_id, date, customer_id, meta
) sales;
```

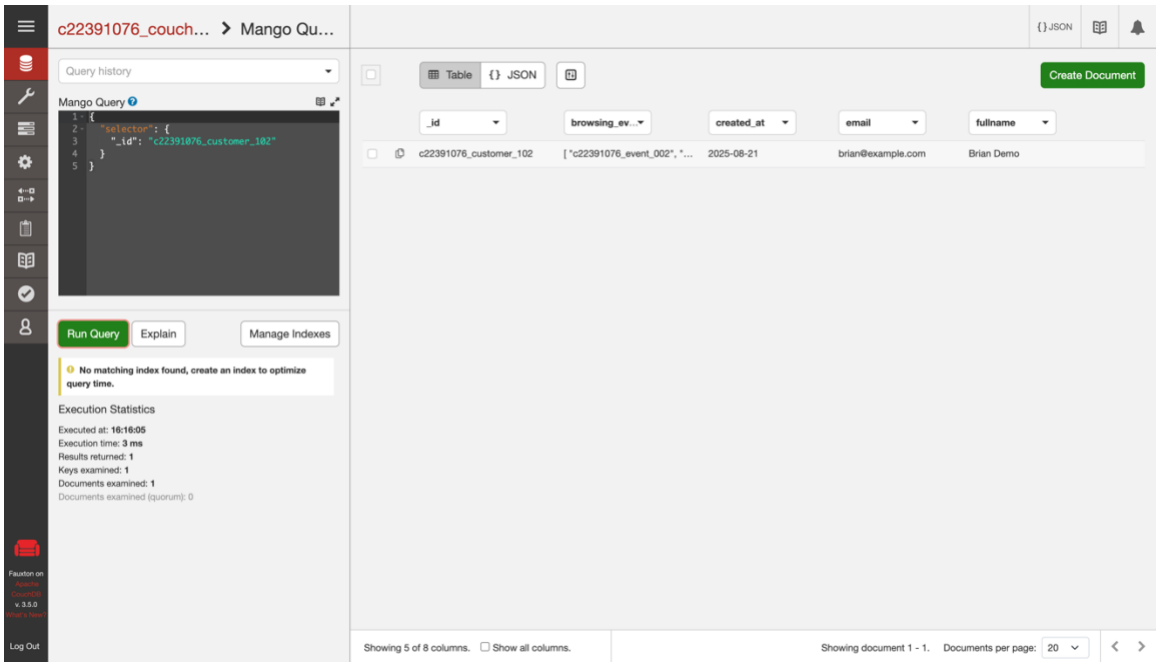
Once exported, JSON files were bulk imported into CouchDB using the _bulk_docs endpoint via curl or the CouchDB Fauxton web UI. This approach streamlines high-volume imports and ensures data integrity.

4.3 Analytical Queries and Views

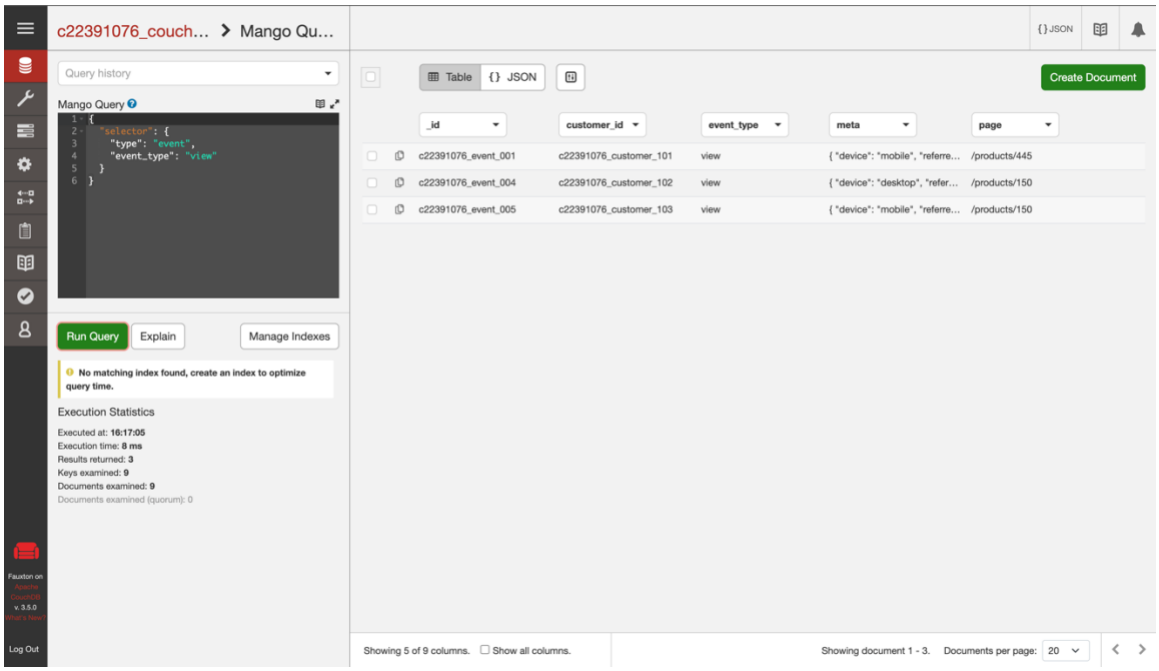
CouchDB supports flexible analytics through both Mango queries and custom MapReduce views.



Shows a Mango query selecting events of type "event" for a specific customer (customer_id: c22391076_customer_102).Purpose: Retrieve all events related to one customer for behavioural analytics.



Query fetching the full customer document by _id (customer lookup or enrichment).



Mango query retrieving all "view" events (page views) for all customers, useful for analysing browsing behaviour.

4.4 Comparison with Operational Relational DB

This section compares how specific analytical queries are constructed in CouchDB (using Mango queries or MapReduce views) versus SQL in PostgreSQL. The following table outlines representative examples:

Analytical Task	CouchDB (Mango/View)	PostgreSQL (SQL)
Get all events for one customer	{ "selector": { "customer_id": "c22391076_customer_102", "type": "event" } }	SELECT * FROM events WHERE customer_id = 'c22391076_customer_102' AND type = 'event';
Lookup customer document by ID	{ "selector": { "_id": "c22391076_customer_102" } }	SELECT * FROM customers WHERE customer_id = 'c22391076_customer_102';
List all product view events	{ "selector": { "type": "event", "event_type": "view" } }	SELECT * FROM events WHERE type = 'event' AND event_type = 'view';

Summary of evidence: See included screenshots for Mango query execution in CouchDB through Fauxton.

Brief Conclusion

- **Speed:** For indexed and simple structure data, PostgreSQL typically queries faster (especially on large datasets) due to query planner optimizations. CouchDB queries remain responsive for moderate dataset sizes but can be slower without proper indexing, as indicated by the "create an index to optimize query time" notice.
- **Flexibility:** CouchDB excels at querying varied, evolving, or semi-structured data, particularly through the use of meta fields and JSON-based queries, while PostgreSQL enforces stricter schema but benefits from robust JOIN and aggregate features.
- **Ease of Querying:** Mango queries are approachable for straightforward document searches and filtering, requiring knowledge of JSON structure rather than SQL. However, complex aggregations are simpler to express (and often faster to execute) in SQL.

Both platforms have strengths suited to particular analytical needs: CouchDB for schema-on-read and exploratory user analytics, and PostgreSQL for performance-intensive, complex reporting.

5. Comparative Discussion (PostgreSQL vs CouchDB)

PostgreSQL and CouchDB each offer distinct strengths and are best leveraged together for a hybrid analytics architecture. Direct comparison across main dimensions highlights why both were used.

Query Capabilities:
PostgreSQL supports powerful, expressive SQL queries, including complex JOINs, nested subqueries, and window functions. It excels at relational analytics and integrates well with established BI/reporting tools. CouchDB, using Mango queries and MapReduce views, offers straightforward filtering and flexible aggregation on semi-structured JSON documents. While Mango is excellent for searches and filtered analytics (e.g. quickly finding events for a customer), it lacks the advanced relational features of SQL.

Performance Scalability:
With proper indexing, PostgreSQL scales comfortably to millions of rows and can execute heavy workloads efficiently, especially for structured data. CouchDB scales horizontally with clustering and is resilient to partial outages, but Mango and MapReduce performance on very large datasets may lag unless indexes are well designed and queries avoid cross-document relations.

Schema Rigidity vs Flexibility:
PostgreSQL enforces a strict schema, ensuring data integrity but requiring explicit migrations for any change. CouchDB enables a flexible, dynamic document structure: new fields or nested data are added without schema migrations, perfectly supporting evolving analytics requirements and semi-structured “meta” fields. This flexibility encourages experimentation and rapid iteration, which is ideal for new user engagement or behavioural analytics.

CouchDB as Complement, Not Replacement:
CouchDB is used alongside the DW, not as a replacement. It provides an agile layer for ingesting, storing, and querying heterogeneous or fast-changing data, such as user events, logs, or meta attributes, that would be awkward or slow to shoehorn into a rigid relational design. Most aggregations, historical reports, and dimensional analysis continue to run efficiently on the warehouse, while CouchDB empowers quick, exploratory, or non-traditional analysis.

Hybrid Strengths:
Leveraging both systems ensures the project benefits from mature, high-performance relational analytics and cutting-edge flexibility. The hybrid model supports advanced reporting and operational BI, while facilitating innovation in semi-structured and real-time analytics.

Both systems enhance each other, resulting in an end-to-end analytics solution that’s robust but never static, ready for new data, new questions, and future growth.

6. Critical Reflection and Conclusion

This project highlighted the balance between structure and flexibility, and the real-world trade-offs encountered in hybrid data warehousing. One of the central design decisions was when to utilize the strict schemas and optimized performance of PostgreSQL versus the agility and dynamic structure of CouchDB.

Trade-offs:
Optimizing for performance in PostgreSQL with temp tables and indexes delivered clear run-time improvements, particularly for repeatable reporting queries. However, this came at the cost of rigid schema requirements and additional maintenance as new data sources or formats emerged. Meanwhile, shifting semi-structured "meta" data and evolving event attributes to CouchDB provided rapid flexibility, but required careful indexing and yielded lower query speeds on larger volumes, a calculated trade-off between agility and raw speed. CouchDB's lack of native referential integrity meant that data durability and consistency had to be managed with disciplined ETL scripts, not constraints.

Effectiveness of Optimizations:
Evidence from EXPLAIN ANALYZE and query timing confirmed the value of traditional DW optimizations: index creation and temporary staging tables gave substantial speedups. Similarly, early index warnings from CouchDB Mango queries revealed the need for careful planning, even in NoSQL, optimization is essential for scale. The hybrid architecture made it possible to capitalize on strengths from both contexts.

Improvements for Phase 3:
For future phases, greater integration between the DW and CouchDB (such as CDC pipelines or event-driven updates) would be prioritized to streamline consistency. Automated index management, more map-reduce views, and real-time ETL pipelines could further bridge the two worlds. Scaling CouchDB's performance would mean paying closer attention to document size and index planning.

Lessons Learned:
The hybrid model is not a cop-out or a mere workaround; it's a strength. Design should be guided by the types of queries, analytics required, and rate of schema evolution, using structure when you can, but embracing flexibility when you must. The most effective analytics solution emerges not from choosing one approach, but by strategically combining both.

7. Evidence of Deployment

- This does not contribute to the word count.
    - It can be in separate files(s), which should be clearly named to reflect the content as well as including your student number somewhere in the file name.
    - If you are including the evidence in separate files then in this section you identify where each type of evidence can be located.
- Screenshots + logs of real deployment
  Must show:
  – docker ps
  – PostgreSQL timestamps (select now()) + EXPLAIN ANALYZE output
  – CouchDB root endpoint JSON, query results, and design documents
- Any log timestamps or verification of query execution
  – Databases include your student number
- Tip: Number and label all evidence files clearly.
- Ensure all database schemas/names and file names include your student number and you show this as part of your evidence of query results/documents etc.
    - **Note**: You do not need to rename the rel_src schema to include your student number but you can if you wish to. You do need to rename the dw_lite schema. You can rename a schema using the ALTER command : ALTER SCHEMA current_schema_name RENAME TO new_schema_name;

Figure 7.1: Docker Container Status

Figure 7.2: PostgreSQL Timestamp Proof

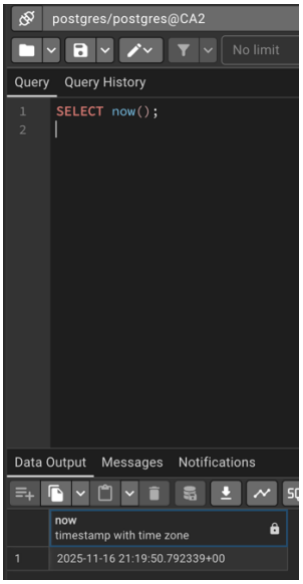


Figure 7.3: EXPLAIN ANALYZE Query Output

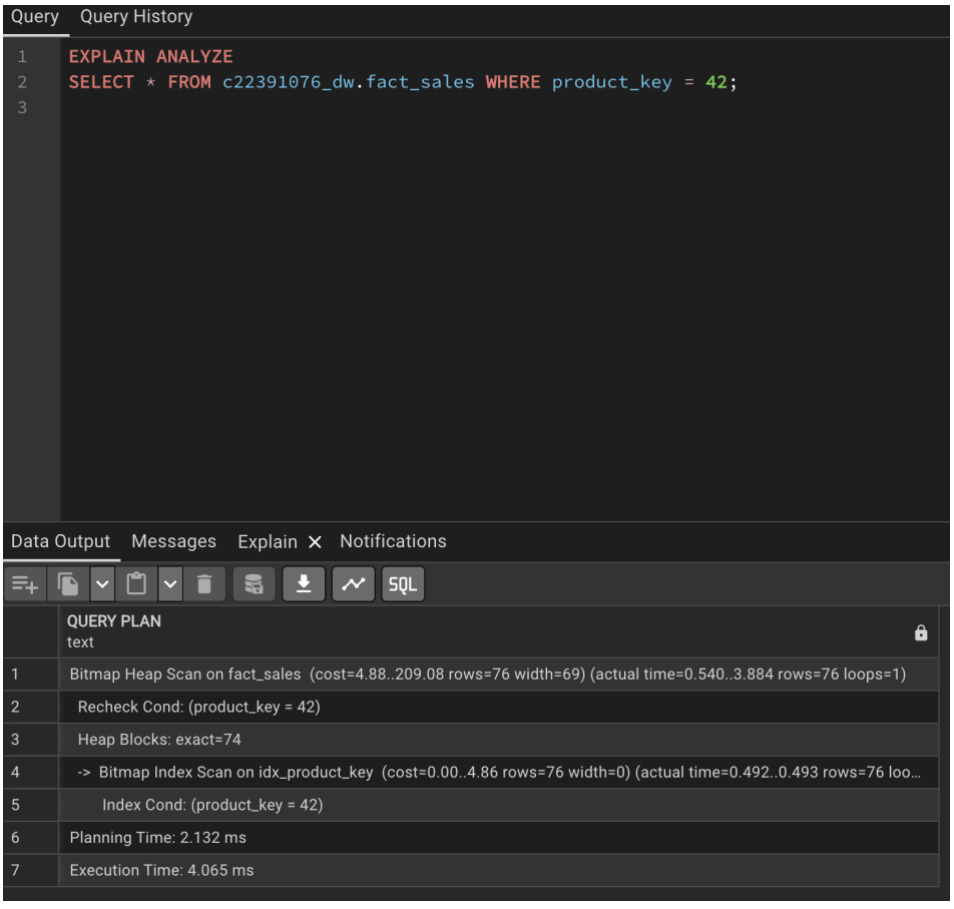


Figure 7.4: CouchDB Root Endpoint

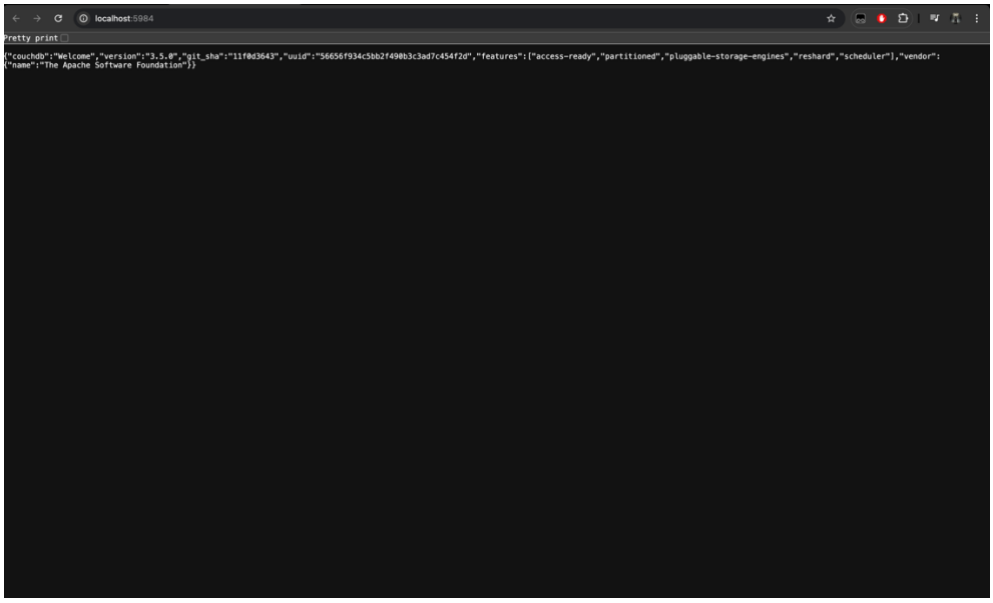


Figure 7.5: CouchDB Database and Design/Document List

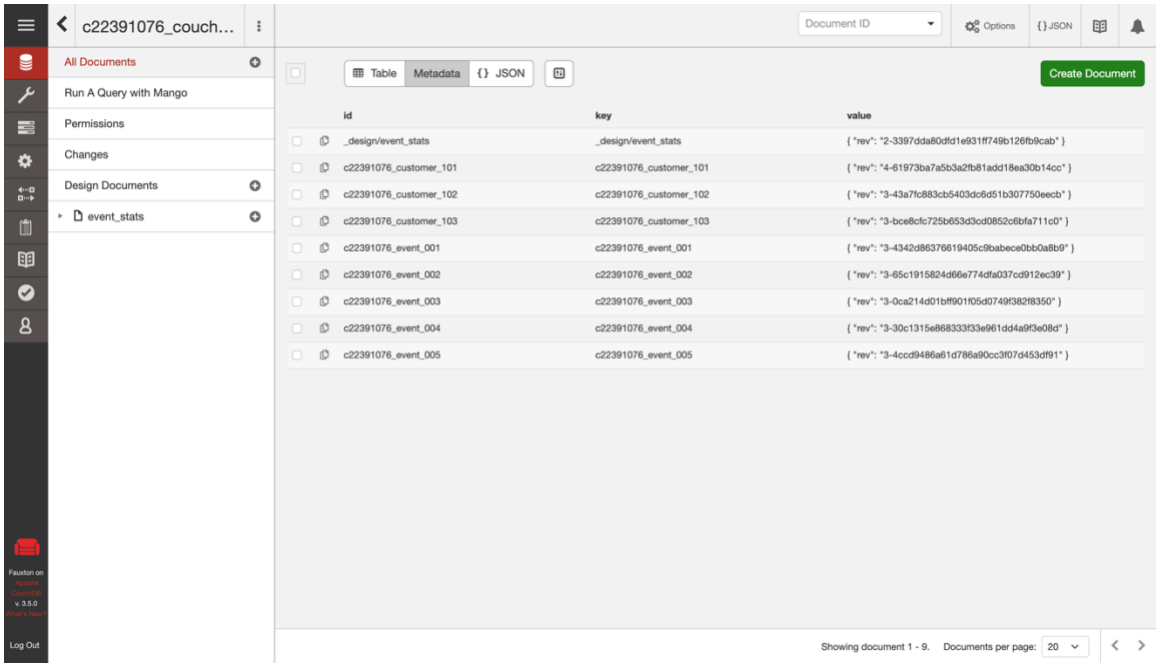
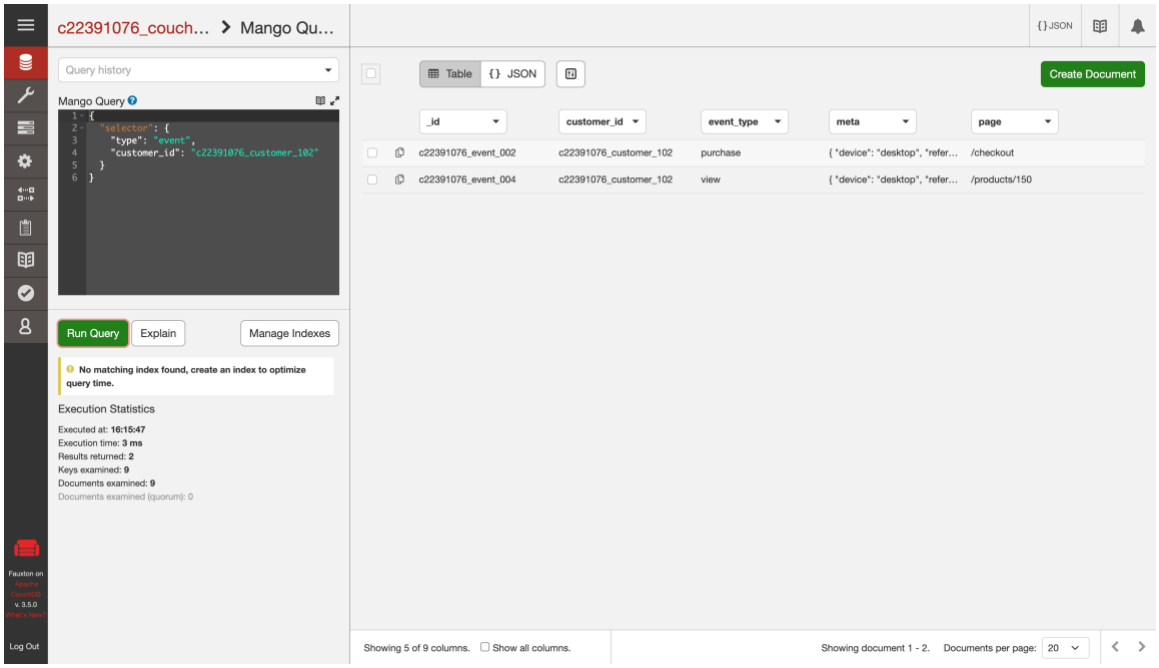
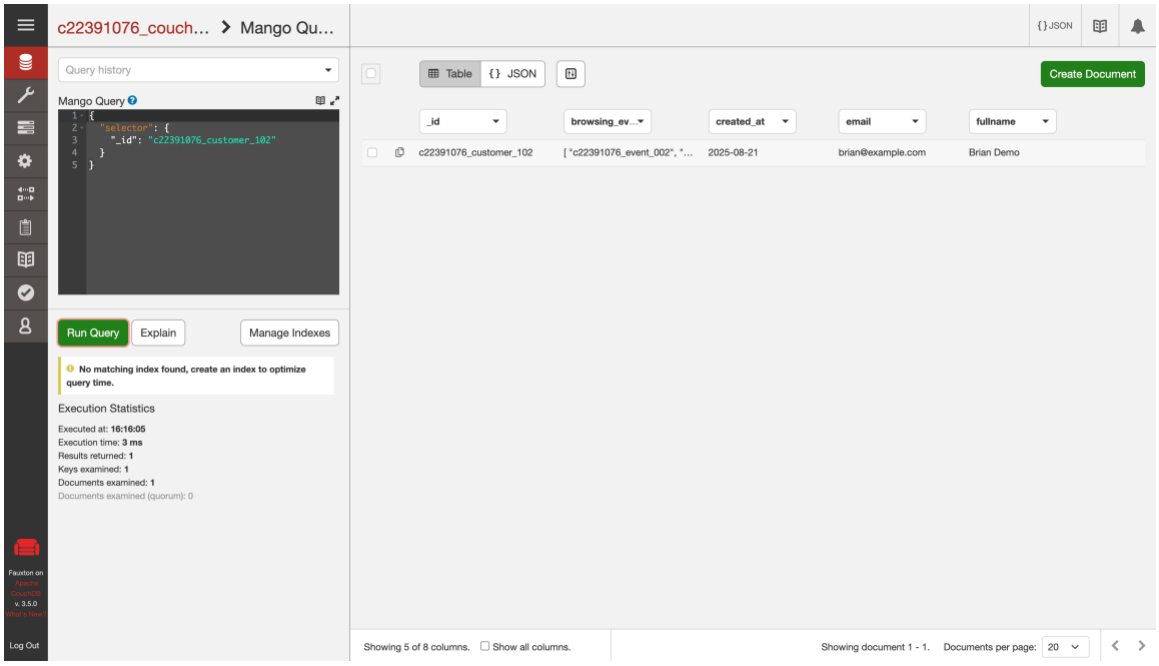


Figure 7.6+: Mango Query and Results Evidence


c22391076_couch... > Mango Qu...
{} JSON
Query history
Mango Query
{
"selector": {
"type": "event",
"customer_id": "c22391076_customer_102"
}
}
Table
{} JSON
Create Document
_id
customer_id
event_type
meta
page
c22391076_event_002
c22391076_customer_102
purchase
{ "device": "desktop", "refer...
/checkout
c22391076_event_004
c22391076_customer_102
view
{ "device": "desktop", "refer...
/products/150
Run Query
Explain
Manage Indexes
No matching index found, create an index to optimize query time.
Execution Statistics
Executed at: 16:15:47
Execution time: 3 ms
Results returned: 2
Keys examined: 9
Documents examined: 9
Documents examined (quorum): 0
Fauxton on Apache CouchDB v. 3.5.0
What's New?
Log Out
Showing 5 of 9 columns.
Show all columns.
Showing document 1 - 2.
Documents per page:
20


c22391076_couch... > Mango Qu...
{} JSON
Query history
Mango Query
{
"selector": {
"_id": "c22391076_customer_102"
}
}
Table
{} JSON
Create Document
_id
browsing_ev...
created_at
email
fullname
c22391076_customer_102
["c22391076_event_002", "...
2025-08-21
brian@example.com
Brian Demo
Run Query
Explain
Manage Indexes
No matching index found, create an index to optimize query time.
Execution Statistics
Executed at: 16:16:05
Execution time: 3 ms
Results returned: 1
Keys examined: 1
Documents examined: 1
Documents examined (quorum): 0
Fauxton on Apache CouchDB v. 3.5.0
What's New?
Log Out
Showing 5 of 8 columns.
Show all columns.
Showing document 1 - 1.
Documents per page:
20

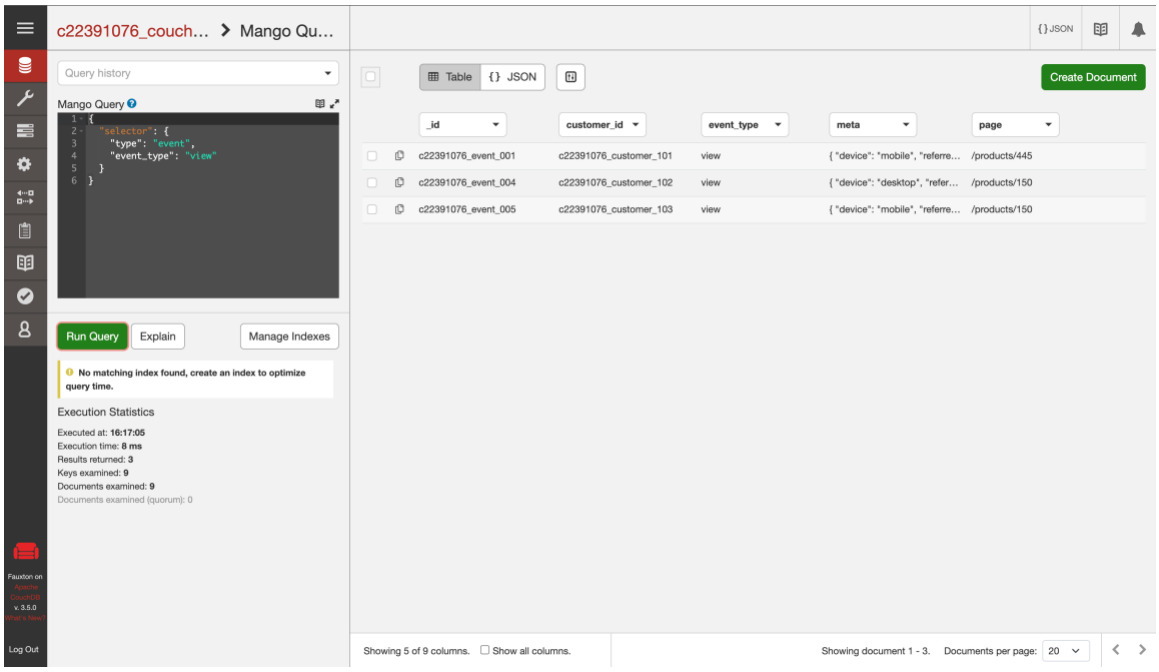


Figure 7.7: CouchDB curl export

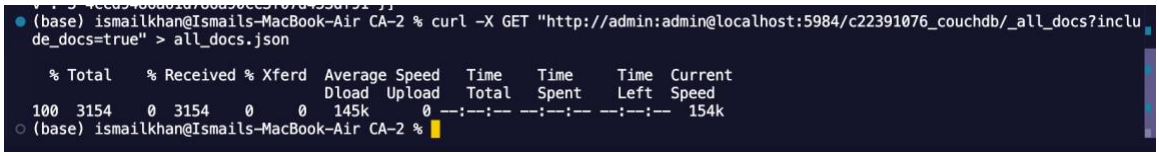


Figure 7.8: CouchDB curl Bulk import

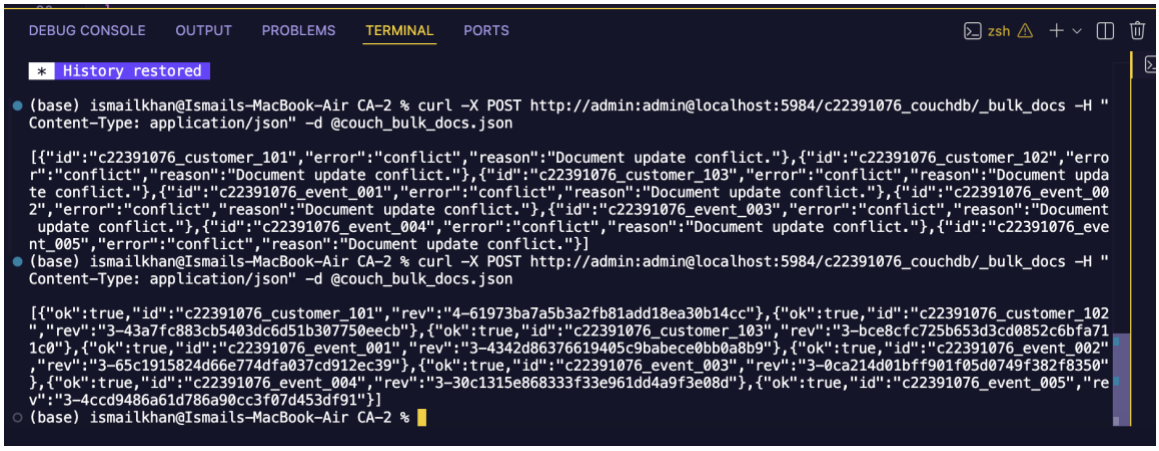
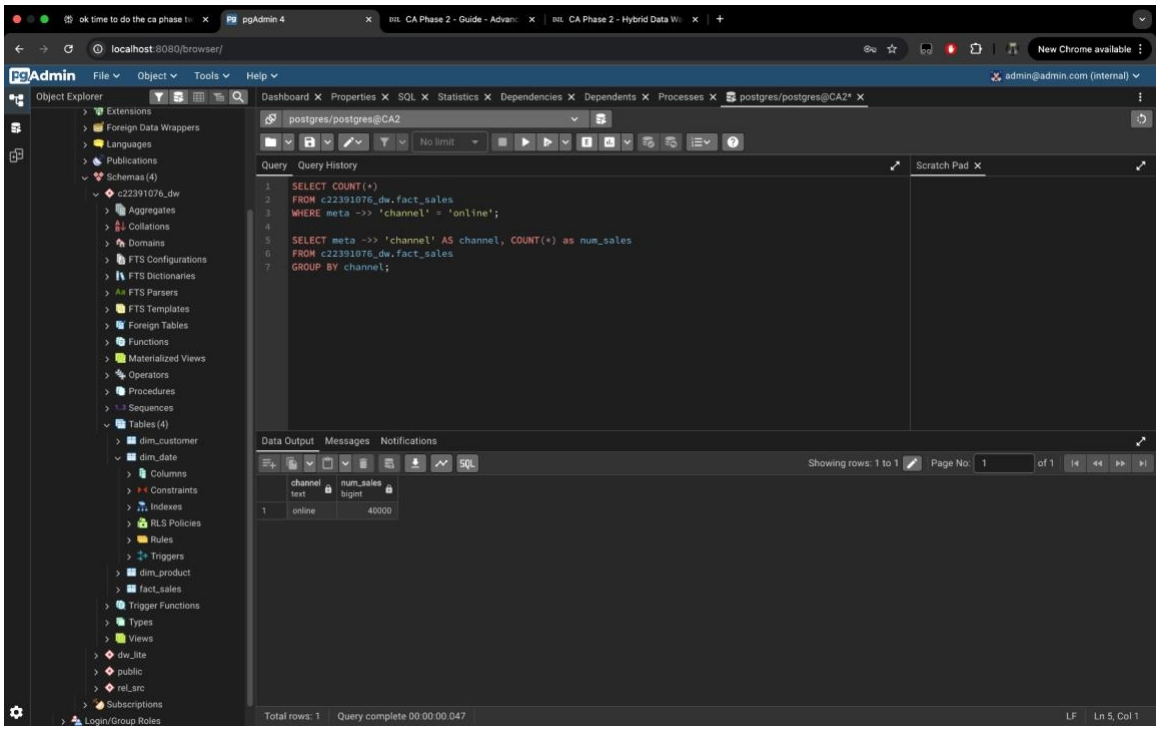
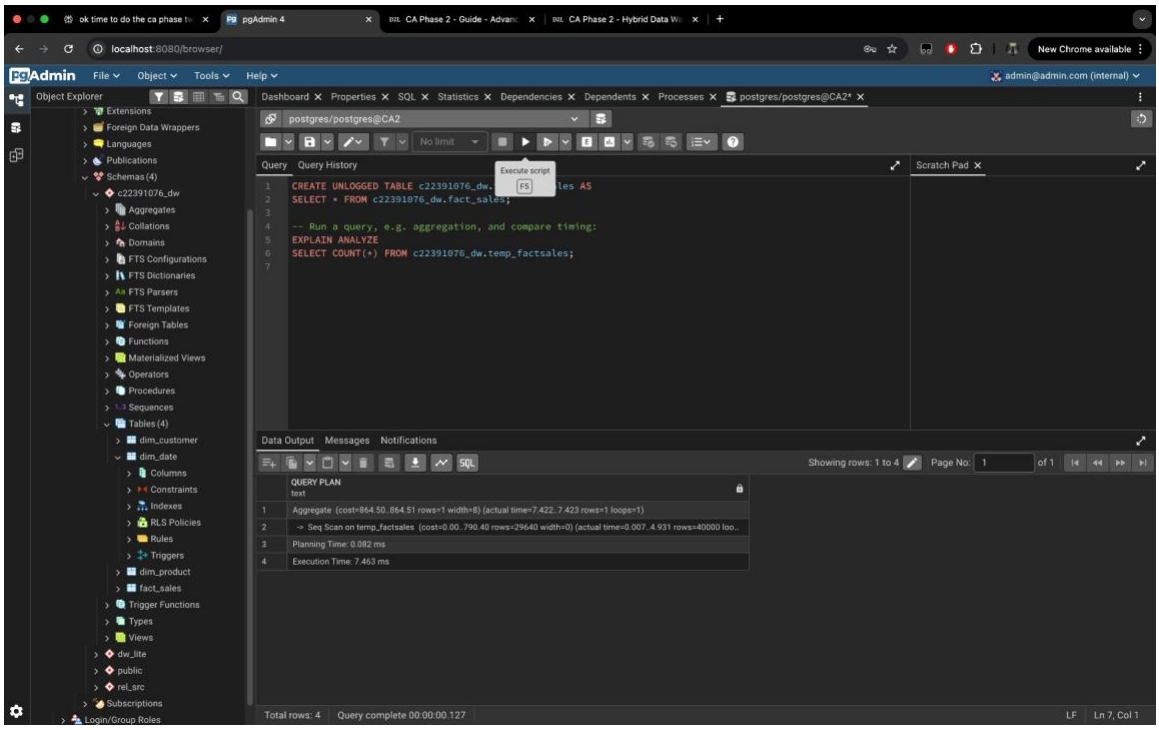
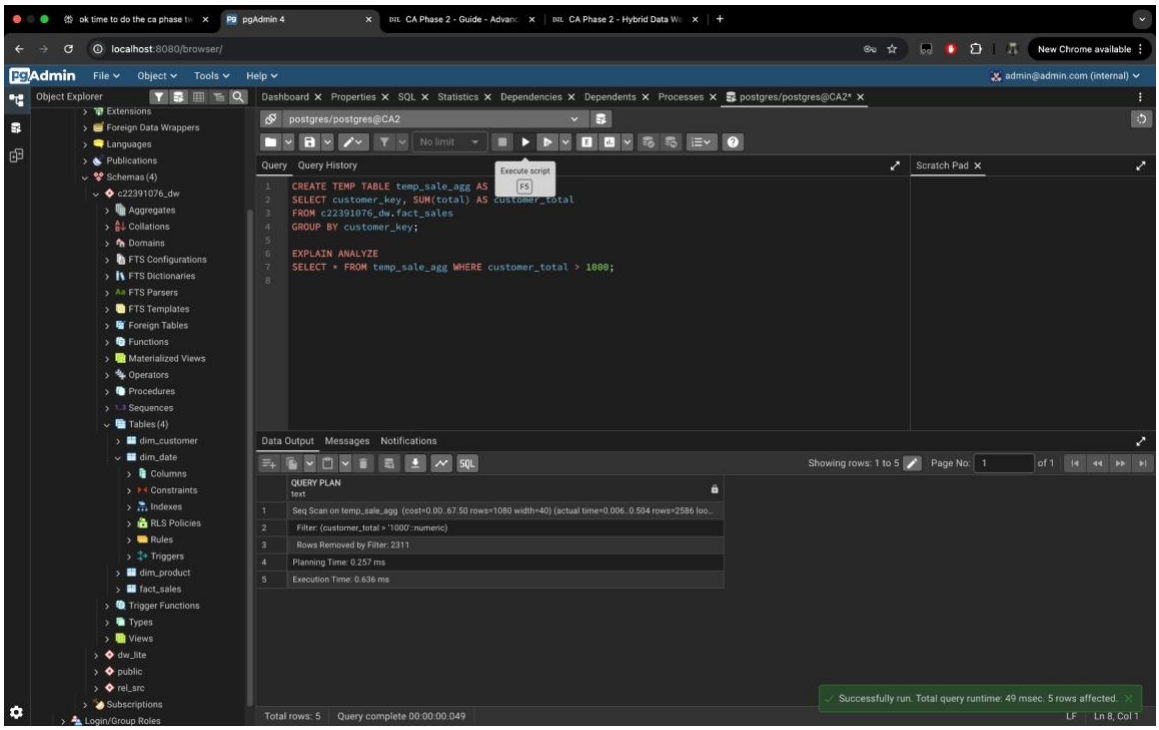
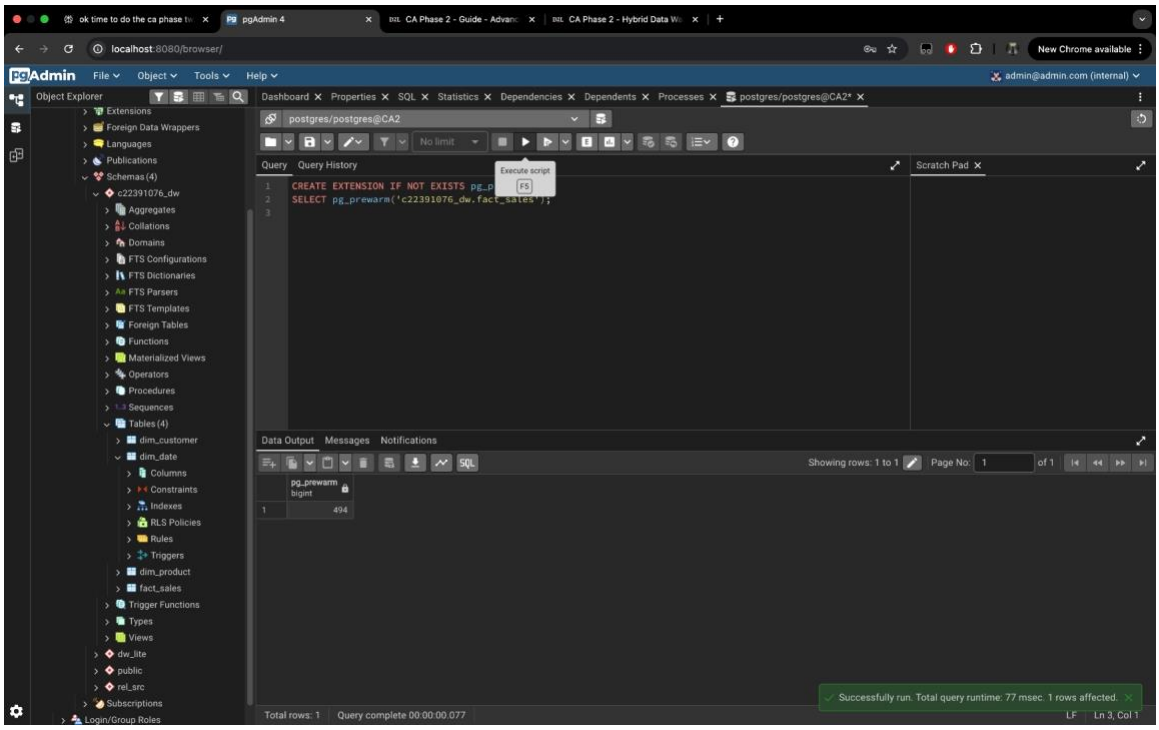


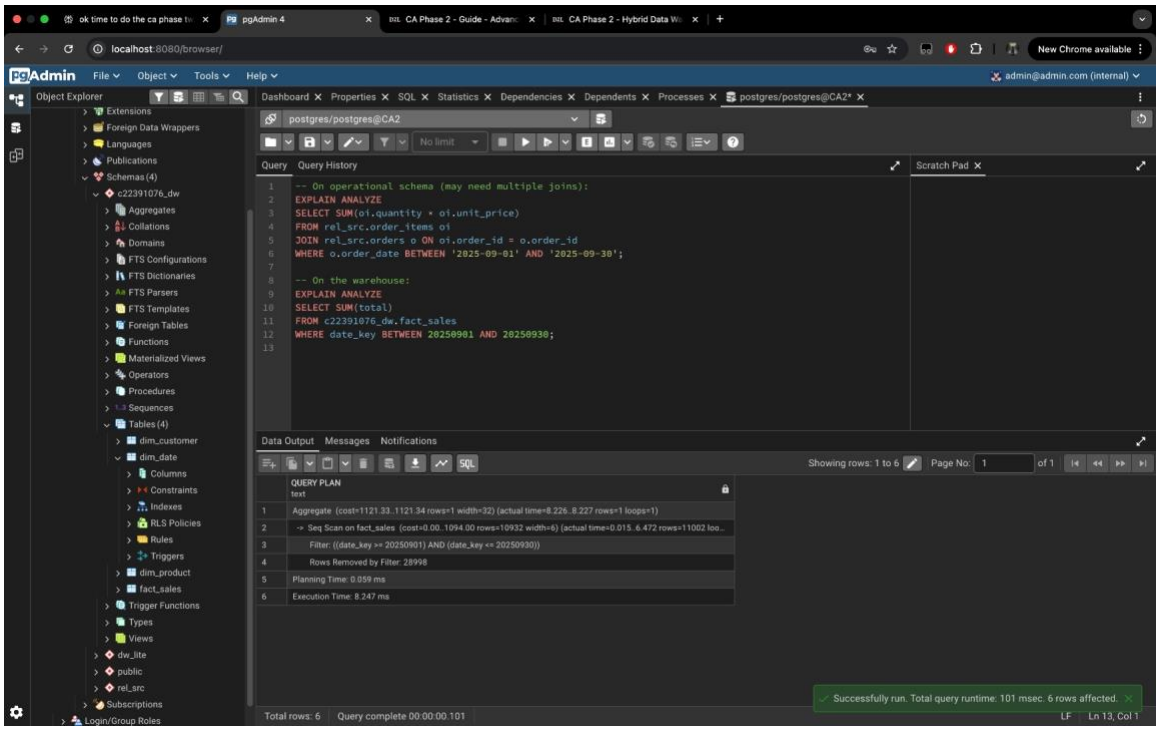
Figure 7.9: Log/Other System Timestamps

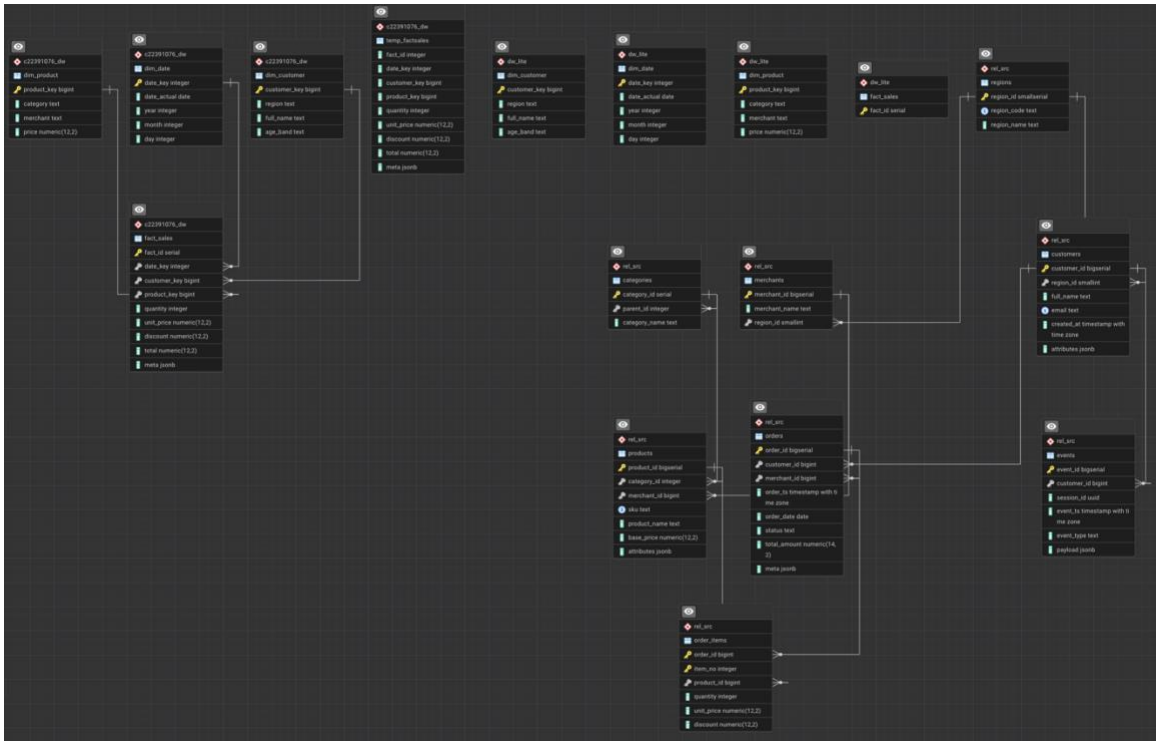



pgAdmin 4
localhost:8080/browser/
postgres/postgres@CA2
CREATE UNLOGGED TABLE c22391076_dw.
SELECT * FROM c22391076_dw.fact_sales;
-- Run a query, e.g. aggregation, and compare timing:
EXPLAIN ANALYZE
SELECT COUNT(*) FROM c22391076_dw.temp_factsales;
QUERY PLAN
Aggregate (cost=864.50..864.51 rows=1 width=8) (actual time=7.422..7.423 rows=1 loops=1)
-> Seq Scan on temp_factsales (cost=0.00..790.40 rows=29640 width=0) (actual time=0.007..4.931 rows=40000 loo..
Planning Time: 0.082 ms
Execution Time: 7.463 ms
Total rows: 4
Query complete 00:00:00.127


pgAdmin 4
localhost:8080/browser/
postgres/postgres@CA2
CREATE TEMP TABLE temp_sale_agg AS
SELECT customer_key, SUM(total) AS customer_total
FROM c22391076_dw.fact_sales
GROUP BY customer_key;
EXPLAIN ANALYZE
SELECT * FROM temp_sale_agg WHERE customer_total > 1000;
QUERY PLAN
Seq Scan on temp_sale_agg (cost=0.00..67.50 rows=1080 width=40) (actual time=0.006..0.504 rows=2586 loo..
Filter: (customer_total > '1000'::numeric)
Rows Removed by Filter: 2311
Planning Time: 0.257 ms
Execution Time: 0.636 ms
Successfully run. Total query runtime: 49 msec. 5 rows affected.
Total rows: 5
Query complete 00:00:00.049


pgAdmin 4
localhost:8080/browser/
CREATE EXTENSION IF NOT EXISTS pg_p
SELECT pg_prewarm('c22391076_dw.fact_sales');
Data Output
Messages
Notifications
pg_prewarm
bigint
494
Successfully run. Total query runtime: 77 msec. 1 rows affected.
Total rows: 1
Query complete 00:00:00.077


pgAdmin 4
localhost:8080/browser/
-- On operational schema (may need multiple joins):
EXPLAIN ANALYZE
SELECT SUM(oi.quantity * oi.unit_price)
FROM rel_src.order_items oi
JOIN rel_src.orders o ON oi.order_id = o.order_id
WHERE o.order_date BETWEEN '2025-09-01' AND '2025-09-30';
-- On the warehouse:
EXPLAIN ANALYZE
SELECT SUM(total)
FROM c22391076_dw.fact_sales
WHERE date_key BETWEEN 20250901 AND 20250930;
Data Output
Messages
Notifications
QUERY PLAN
text
Aggregate (cost=1121.33..1121.34 rows=1 width=32) (actual time=8.226..8.227 rows=1 loops=1)
-> Seq Scan on fact_sales (cost=0.00..1094.00 rows=10932 width=6) (actual time=0.015..6.472 rows=11002 loo...
Filter: ((date_key >= 20250901) AND (date_key <= 20250930))
Rows Removed by Filter: 28998
Planning Time: 0.059 ms
Execution Time: 8.247 ms
Successfully run. Total query runtime: 101 msec. 6 rows affected.
Total rows: 6
Query complete 00:00:00.101


c22391076_dw
dim_product
product_key bigint
category text
merchant text
price numeric(12,2)
c22391076_dw
dim_date
date_key integer
date_actual date
year integer
month integer
day integer
c22391076_dw
dim_customer
customer_key bigint
region text
full_name text
age_band text
c22391076_dw
fact_sales
fact_id serial
date_key integer
customer_key bigint
product_key bigint
quantity integer
unit_price numeric(12,2)
discount numeric(12,2)
total numeric(12,2)
meta jsonb
c22391076_dw
temp_factsales
fact_id integer
date_key integer
customer_key bigint
product_key bigint
quantity integer
unit_price numeric(12,2)
discount numeric(12,2)
total numeric(12,2)
meta jsonb
dw_lite
dim_customer
customer_key bigint
region text
full_name text
age_band text
dw_lite
dim_date
date_key integer
date_actual date
year integer
month integer
day integer
dw_lite
dim_product
product_key bigint
category text
merchant text
price numeric(12,2)
dw_lite
fact_sales
fact_id serial
rel_src
regions
region_id smallserial
region_code text
region_name text
rel_src
categories
category_id serial
parent_id integer
category_name text
rel_src
merchants
merchant_id bigserial
merchant_name text
region_id smallint
rel_src
customers
customer_id bigserial
region_id smallint
full_name text
email text
created_at timestamp with time zone
attributes jsonb
rel_src
products
product_id bigserial
category_id integer
merchant_id bigint
sku text
product_name text
base_price numeric(12,2)
attributes jsonb
rel_src
orders
order_id bigserial
customer_id bigint
merchant_id bigint
order_ts timestamp with time zone
order_date date
status text
total_amount numeric(14,2)
meta jsonb
rel_src
events
event_id bigserial
customer_id bigint
session_id uuid
event_ts timestamp with time zone
event_type text
payload jsonb
rel_src
order_items
order_id bigint
item_no integer
product_id bigint
quantity integer
unit_price numeric(12,2)
discount numeric(12,2)