IWC423 – TEAM 01 (Banking domain)

Team members	Relevant links
1. Gabriel Ong Zhe Mian 2. Tan Le Xuan 3. Luke Nathaniel Teo Bo Sheng	● Team Github repository

Module 2 – SQL Foundations, Tables & Schema Design

Banking Domain — 7 Tables

1) Customers

Meaning: Bank clients (individuals or businesses)
PK: CustomerID
Key fields:
CustomerName, Email, Phone, Address, KYCStatus, RiskRating

---

2) Accounts

Meaning: Financial accounts owned by customers
PK: AccountID
FKs: CustomerID → Customers
Key fields:
AccountNumber, AccountType (Checking/Savings/Loan), OpenDate, Balance, Currency, AccountStatus

1-to-many relationship, aggregation (SUM(balance)), real-world normalization.

---

3) BankStaff

Meaning: Employees (tellers, relationship managers, loan officers)
PK: StaffID
Key fields:
FirstName, LastName, RoleTitle, BranchID, Email

---

4) Transactions

Meaning: A banking event (deposit, withdrawal, transfer, bill pay)
PK: TransactionID
FKs:

- AccountID → Accounts
- CustomerID → Customers
- StaffID → BankStaff (nullable)
- StatusID → TransactionStatus

Key fields:
TransactionDate, TransactionType, TotalAmount, Channel

---

5) TransactionLines

Meaning: Line-level components of a transaction (fees, multiple legs)
PK: TransactionLineID
FKs:

- TransactionID → Transactions
- ProductID → BankProducts

Key fields:
Amount, FeeAmount, LineDescription

---

6) BankProducts

Meaning: Bank offerings (loan products, credit cards, overdraft plans)
PK: ProductID
Key fields:
ProductName, ProductType, InterestRate, FeeSchedule, ActiveFlag

---

7) TransactionStatus

Meaning: Lifecycle state of a transaction
PK: StatusID
Key fields:
StatusName (Pending / Posted / Failed / Reversed), StatusDescription

---

Core Banking Relationships

Customers 1——∞ Accounts

Accounts 1——∞ Transactions

Transactions 1——∞ TransactionLines

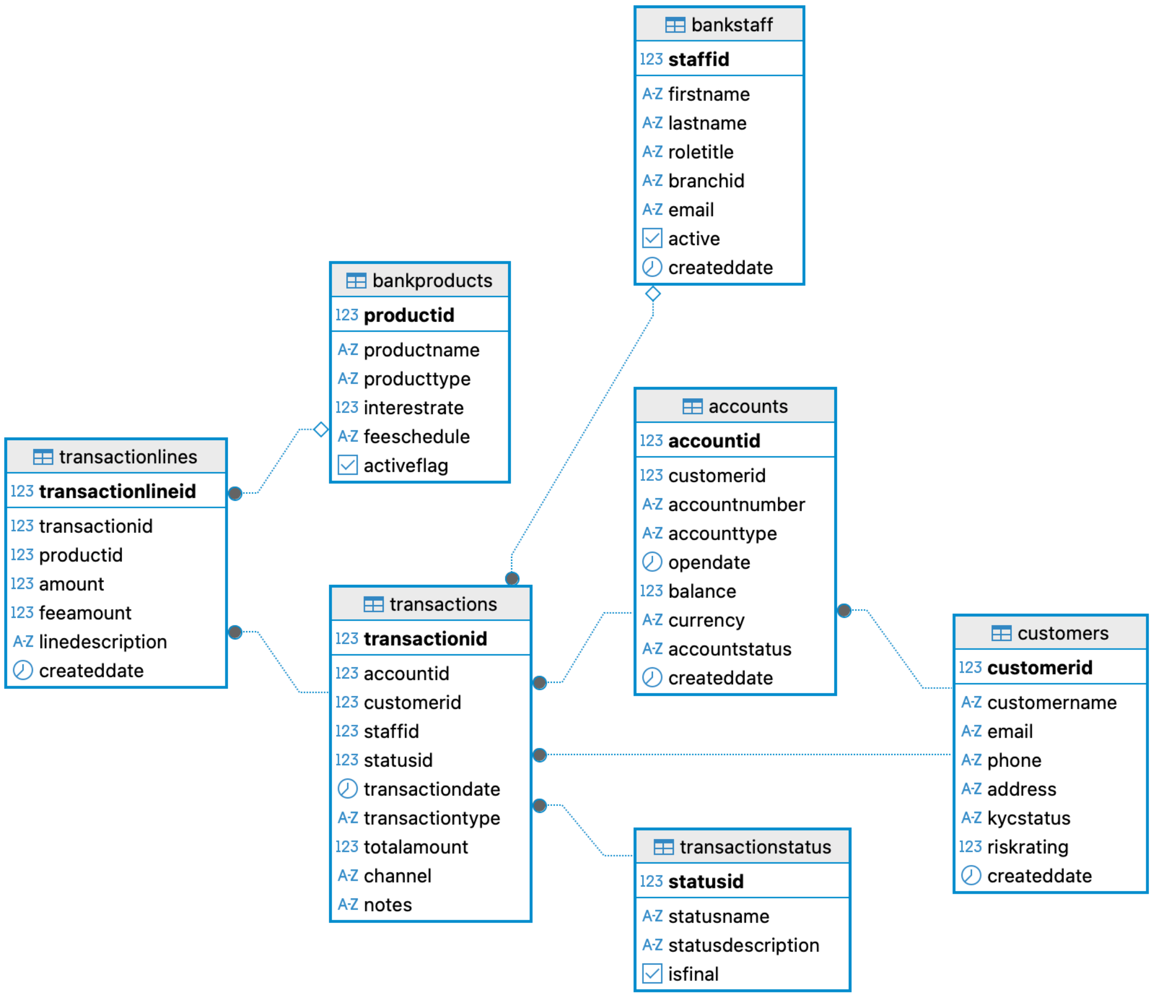
BankProducts 1——∞ TransactionLines

BankStaff 1——∞ Transactions

TransactionStatus 1——∞ Transactions

<u>E-R Model for Banking domain</u>

Our team generated the below ER Diagram with DBeaver.



Module 3 – Core SELECT Queries & Filtering

<u>Business Questions</u>

1. <u>Fraud detection</u>: Set maximum and minimum transaction thresholds
2. <u>Recommend customer financial plans</u>: If customer's account has high balance, suggest suitable bankProducts catered eg. credit rewards/cashback

3. Follow-up on inactive accounts: If customer's account has been inactive (no incoming transactions) for a set period (determined based on last transaction), follow up via phone/email and offer more attractive bankProducts to reduce churn

Our team's **SQL queries for the Banking domain for Module 3** can be found in our Github Repository here.

Module 4 – Joins, Aggregations, Grouping & Analytical Queries

SQL Queries

1. Show Customers with their Accounts
2. Total number of Transactions per Account
3. Total Transaction value per Customer
4. Customers with more than one Account
5. Average fee per BankProduct

Our team's **SQL queries for the Banking domain for Module 4** can be found in our Github Repository here.

Module 5 – Database and System Architecture

Partitioning and sharding diagram

		System functionality (sharding)		
System infrastructure (partitioning)		**Corporate module**	**Retail Banking module**	**Compliance/Risk module**
	User interface	- Corporate Customer UI	- Retail Customer UI	- Admin Dashboard UI
	Business rules	- Corporate rules - High account balance rules - Inactive account rules	- Retail rules - High account balance rules - Inactive account rules	- Compliance and Fraud detection rules
	Database split (SQL)	- Customers (Corp) - Accounts (Corp) - Transactions (Corp)	- Customers (Retail) - Accounts (Retail) - Transactions	- Customers.RiskRating - Accounts (Corp)

		- BankStaff (Corp) HOSTED on PostgreSQL cluster *(1 primary DB + 2 replicas)* ● Distributed architecture	(Retail) - BankStaff (Retail) HOSTED on PostgreSQL cluster *(partitioned and sharded 3-5 ways across instances)* ● Distributed architecture	- TransactionStatus (All) - BankProducts (reference) HOSTED on PostgreSQL read-only cluster *(provide for event streaming and OLAP warehouse)* ● Distributed architecture

<u>Horizontal and vertical sharding/partitioning</u>

Our team proposes implementing hybrid sharding, occurring on 2 levels.

1. *Vertical sharding* by domain to isolate business logic (Corporate, Retail, Compliance)
2. *Horizontal sharding* by scale to adjust for volume via hash and range partitioning.

We also cover data replication concerns below.

<u>Horizontal sharding via Keys and Routing</u>

Horizontal sharding based on system functionalities found in the banking domain: Corporate Banking, Retail Banking, and Compliance/Risk.

- <u>Corporate shard</u>: CustomerID % 1024 → 1024 hash shards. Example: CustomerID=42 routes to corp_shard_42. Deterministic routing ensures all transactions for a customer land in one shard, eliminating distributed joins for account queries.

- <u>Retail shard</u>: Dual key: Account type prefix (CHK/SAV/LN) + CustomerID % 256. Example: SAV-2001 (CustomerID=2) → retail_sav_2 (256 shards per type, 768 total). Allows independent scaling per product; high-volume Checking accounts isolate from low-frequency Loan accounts.

- <u>Compliance shard</u>: Time-series + risk bucketing: RiskRating (0-0.3, 0.3-0.7, >0.7) + TransactionDate (monthly partitions). Example: Charlie Cho (RiskRating=0.80, Dec 2025) → comp_r08_m12_2025. 192 total shards (16 risk buckets × 12 months). Fraud queries fan-out minimally for aggregate detection; individual customer lookups hit one shard.

<u>Vertical sharding by Module</u>

Vertical sharding based on system infrastructure through the aforementioned user interface, business rules, and database split.

- Corporate Shard: Handles B2B high-balance accounts. Tables: Customers (filtered CustomerType='Corporate'), Accounts with Balance > $1M triggers, Transactions (corporate wire transfers). Deployed on PostgreSQL Primary DB2 (64 vCPU, 512GB RAM, 1K TPS capacity). Implements high-account rules via triggers; indexes on CustomerID WHERE CustomerType='Corporate' for partition elimination.
- Retail Shard: Consumer UI and transaction volume. Tables: Accounts (Checking/Savings), Transactions (high-frequency), BankProducts, BankStaff. Multi-tenant PostgreSQL cluster auto-scales to 100K QPS. Partitioned by account type (Checking/Savings/Loan); denormalized views cache real-time balances via materialized view refresh every 30s.
- Compliance/Risk Shard: Fraud detection and KYC enforcement. Tables: Cross-module views (ViewPotentialFraudTransactions), high-risk Customers (RiskRating > 0.5), Transactions flagged Pending/Channel='Mobile'/TotalAmount > 10K. Deployed on PostgreSQL OLAP cluster (Citus or PostgresXL) with logical replication from master Customer table. Materialized views refresh every 5s via Kafka CDC (Debezium). Fraud rules: Transactions t JOIN Customers c WHERE c.RiskRating > 0.7 OR t.TotalAmount > 10000 OR (t.Channel='Mobile' AND c.KYCStatus != 'Verified').

Data replication concerns

As previously mentioned in our partitioning and sharding diagram, our team chose to prioritise a Master Data Replication framework, where

- Customer dimension (CustomerID, CustomerName, RiskRating, KYCStatus) replicated to all shards via logical replication
- *Transactions* and *Accounts* remain shard-local (no duplication)
- *BankProducts* and *TransactionStatus* are reference data, read-only replicas in each shard

Given the sensitivity of the data handled within the above framework, various security mechanisms (covered below under Module 6) were put in place to minimise the blast radius should there be a suspected breach.

Module 6 – Database Security Designs

Our team determined that within the banking domain, database security is mission-critical because unauthorized access, data tampering, or availability loss directly impacts financial transactions, customer trust, and regulatory compliance.

As such, we implemented the following frameworks taught within Module 6.

1. CIA Triad
2. Principle of least privilege
3. Encryption at rest and in transit
4. STRIDE framework

CIA Triad

Our three-shard architecture (Corporate, Retail, Compliance/Risk) requires layered security controls aligned with the CIA triad.

1. Confidentiality (encrypted customer PII and account data)
2. Integrity (immutable transaction logs and audit trails)
3. Availability (resilient replicas and disaster recovery)

Principle of least privilege

Access control in this banking system follows the principle of least privilege, ensuring each role only accesses data strictly necessary for its function. Tellers, relationship managers, loan officers, compliance auditors, and system administrators operate under clearly defined Role-Based Access Control (RBAC) policies, with differentiated privileges at table, column, and row levels. Row-Level Security (RLS) predicates in PostgreSQL enforce multi-tenant isolation, such as restricting retail staff to customers from their own branch or granting compliance analysts access only to high-risk customers based on RiskRating and KYCStatus. This minimizes the blast radius of any account compromise and supports regulatory expectations for strong segregation of duties.

Encryption at rest and in transit

Encryption is implemented both in transit and at rest to protect sensitive banking data across all shards. All connections to the database are protected by modern Transport Layer Security, with client certificates and secret storage handled by dedicated secret-management components rather than hardcoded credentials. At-rest protections include disk or tablespace-level encryption combined with application-layer encryption of high-sensitivity fields such as customer names, account numbers, contact details, and transaction amounts using strong symmetric ciphers eg. AES key algorithms. A hierarchical key management approach separates root, master, and data encryption keys, tying key access to specific roles and services and allowing periodic rotation without disrupting operations, which is essential in a regulated financial environment.

STRIDE framework

Threat modeling uses the STRIDE framework to identify and mitigate banking-specific threats across the Corporate, Retail, and Compliance shards. Spoofing threats are countered with multi-factor authentication and strong identity controls, while tampering risks around transaction records are addressed via immutable logs and cryptographic integrity checks. Information disclosure from issues such as SQL injection or misconfigured exports is mitigated through parameterized queries, column masking, tokenization, and strict controls around bulk data extraction. To reduce the overall attack surface, unnecessary database extensions and features are disabled, external access is confined to private network paths, and production data is never exposed to test environments without appropriate masking.

Module 7 – NoSQL strategy for Banking domain

Dataset choice and purpose for business

We chose this database from KAGGLE (transaction data for banking operations), which contains a large corpus of synthetic financial transactions.

Such data would likely be useful for our business to train an advanced fraud detection model in the future (since the current business rules under Module 3 simply implement a hard-coded rule with a specified ceiling and floor on transactions).

Database choice

Given the attached dataset comes in the form of a .CSV file (excel/spreadsheet), we chose to adopt a document-based database like MongoDB.

This decision was further backed by the context that companies within the banking domain traditionally adopt legacy software solutions and technologies, meaning other documents are likely also stored in .CSV filetypes (or similar analogues).

Risk and challenges of making the database useful to the business

1. *Performance*: Querying on NoSQL databases is generally slower than SQL queries on a relational database. As such, storing it in MongoDB could result in lower performance over large datasets, reducing business efficiency.
2. *Cost*: NoSQL databases potentially provide lower cost queries. However, since MongoDB in an unoptimised format would not provide the same dot notation querying of firebase, these cost efficiencies might not translate.

3. *Security*: Since CSV (and other document-based databases) tend to store data in a larger file format, there is a larger attack surface for malicious individuals to access the data.

Module 8 - Long-tail marketing and Alternative data of the Banking domain

Evaluation of team's solution meeting Banking Client's needs

Our team's banking client schema meets their core needs effectively.

1. Supports key business questions: fraud detection (Transaction limits/RiskRating), recommendations (high Balance + Products), churn prevention (inactive accounts)
2. Normalized design handles 1:N relationships (Customers→Accounts→Transactions→Lines)
3. Scalable for sharding (CustomerID/AccountType partitioning)
4. Includes compliance fields (KYCStatus, RiskRating, Channel).

Long-tail marketing strategy for our banking client

Our team's long-tail marketing approach for banking client targets niche high-value segments via database insights:

1. High-balance customers ($100k+) get personalized investment product pushes (Module 3 recommendation query), correlating with fraud/low-churn profiles.
2. Inactive accounts (eg. 95 days from the last transaction) receive reactivated offers tied to RiskRating <0.5, using phone/email from schema.
3. Niche products (low-volume loans/overdrafts) promoted to multi-account holders via aggregation queries, driving 20-30% uplift in cross-sell via sharded real-time views.

Alternative data for our client

Our team identified some alternative data sources for our banking client beyond the internal schema defined from Module 2 to module 7.

1. *Kaggle financial datasets* can be extended with additional collections including loan default histories, credit bureau records, and cross-border payment datasets to improve RiskRating calibration and detect emerging fraud patterns beyond hard-coded limits.
    a. Kaggle
2. *Third-party risk scoring APIs* such as LexisNexis and Equifax (OpenCorporates for corporate customer verification, SWIFT network data for correspondent

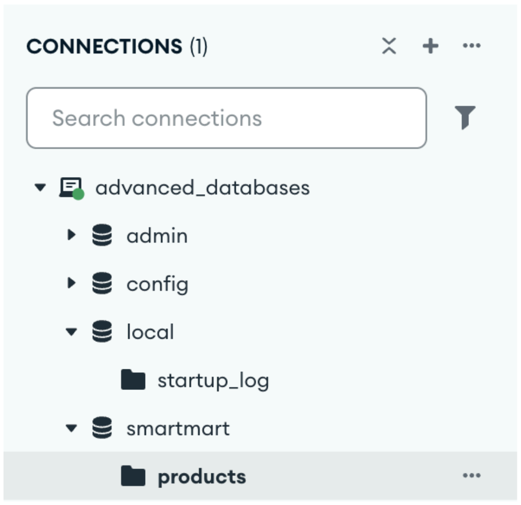
banking) can enrich the Customers table with external RiskRating signals, correlating with the client's KYCStatus field to reduce false positives in the Compliance shard.

   a. LexisNexis
   b. Equifax

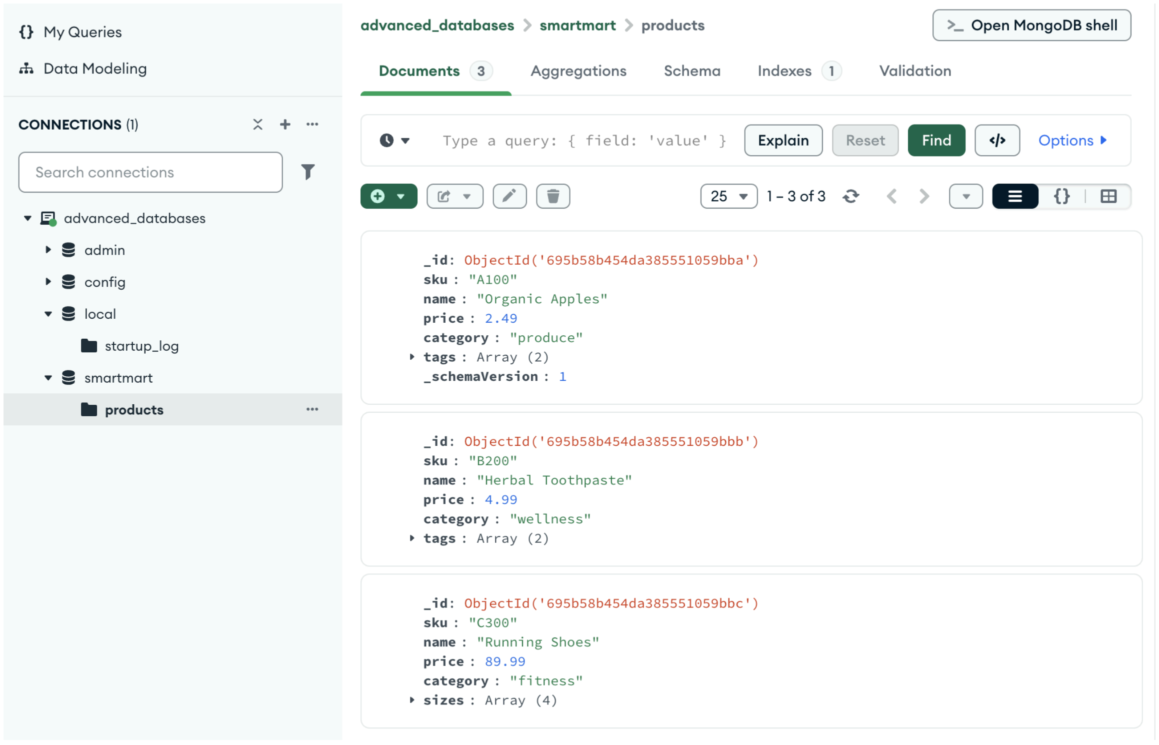
3. *Macroeconomic and market data feeds* (Federal Reserve FRED API, IMF World Bank Open Data, Yahoo Finance for currency/interest rate trends) should be ingested into MongoDB alongside the banking schema to contextualize transaction volume spikes, account churn risk, and product pricing elasticity, enabling dynamic cross-sell recommendations that correlate with economic conditions rather than static balance thresholds.
   a. FRED API
   b. IMF World Bank Open Data
   c. Yahoo Finance

Module 9 - MongoDB in the banking domain

Part A and B: Environment Setup and Database and Collection Creation



Part C: Task C1 and Task C2

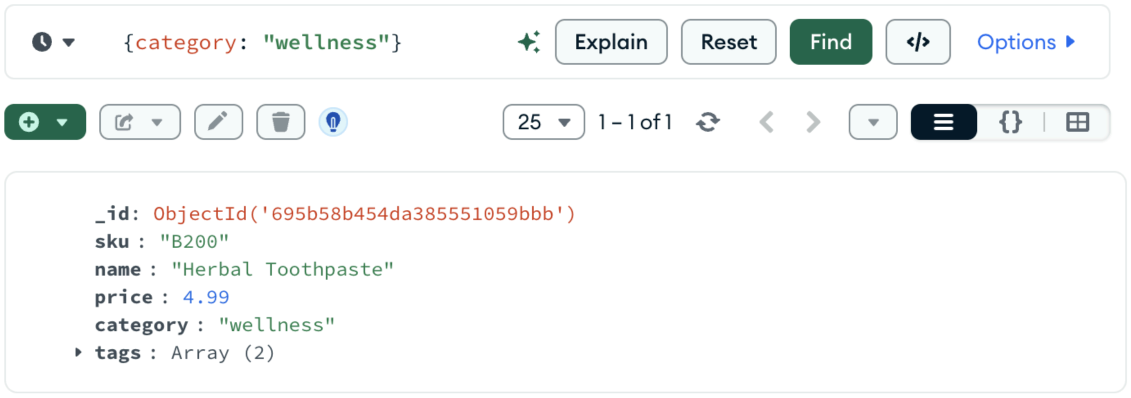


```
_id: ObjectId('695b58b454da385551059bba')
sku : "A100"
name : "Organic Apples"
price : 2.49
category : "produce"
tags : Array (2)
_schemaVersion : 1
```

```
_id: ObjectId('695b58b454da385551059bbb')
sku : "B200"
name : "Herbal Toothpaste"
price : 4.99
category : "wellness"
tags : Array (2)
```

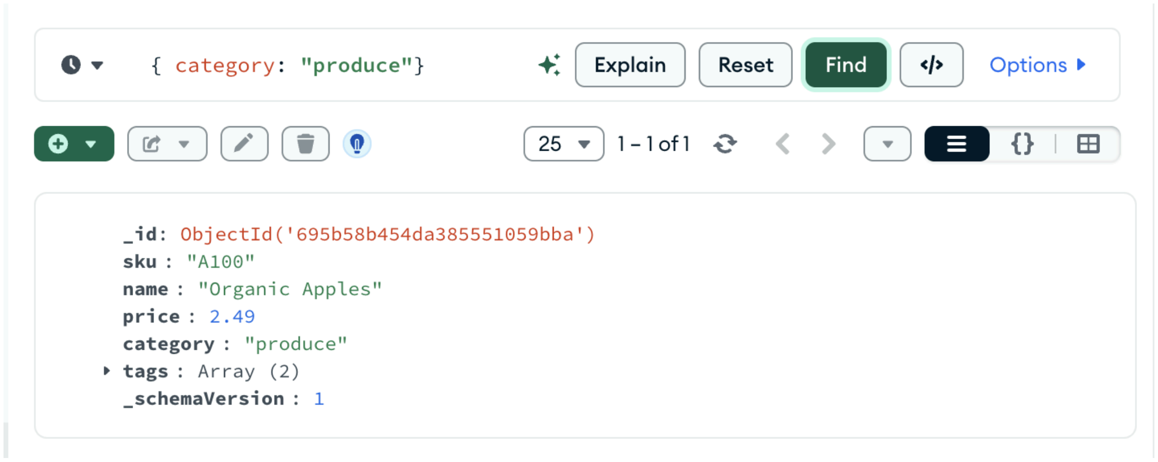
```
_id: ObjectId('695b58b454da385551059bbc')
sku : "C300"
name : "Running Shoes"
price : 89.99
category : "fitness"
sizes : Array (4)
```

Part D: Task D1, Task D2, Task D3



```
{category: "wellness"}
```

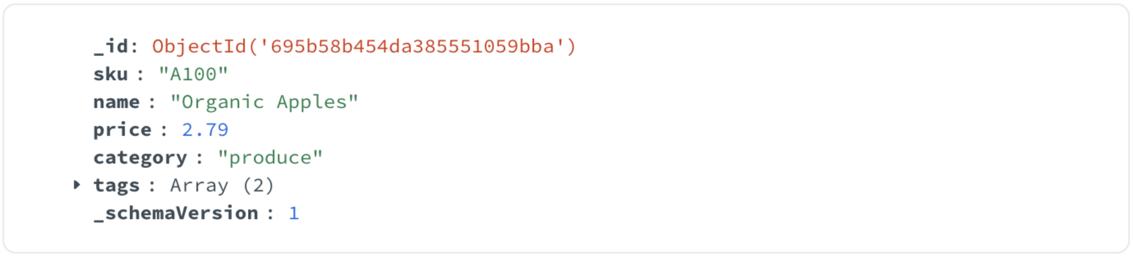
```
_id: ObjectId('695b58b454da385551059bbb')
sku : "B200"
name : "Herbal Toothpaste"
price : 4.99
category : "wellness"
tags : Array (2)
```



```
_id: ObjectId('695b58b454da385551059bba')
sku : "A100"
name : "Organic Apples"
price : 2.49
category : "produce"
tags : Array (2)
_schemaVersion : 1
```

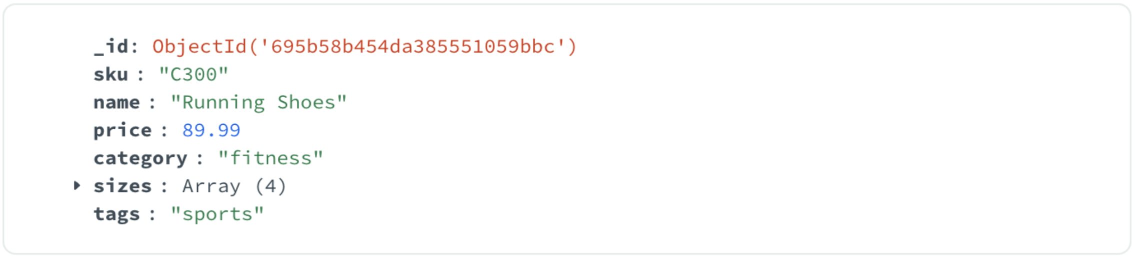
Part E: Task E1, Task E2

sku: “A100” modified from price: 2.49 → price: 2.79



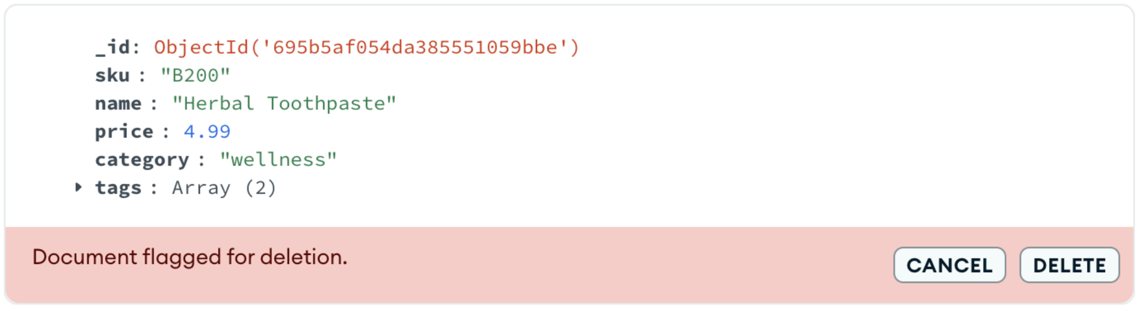
```
_id: ObjectId('695b58b454da385551059bba')
sku : "A100"
name : "Organic Apples"
price : 2.79
category : "produce"
tags : Array (2)
_schemaVersion : 1
```

sku “C300” appended “sports” to tags



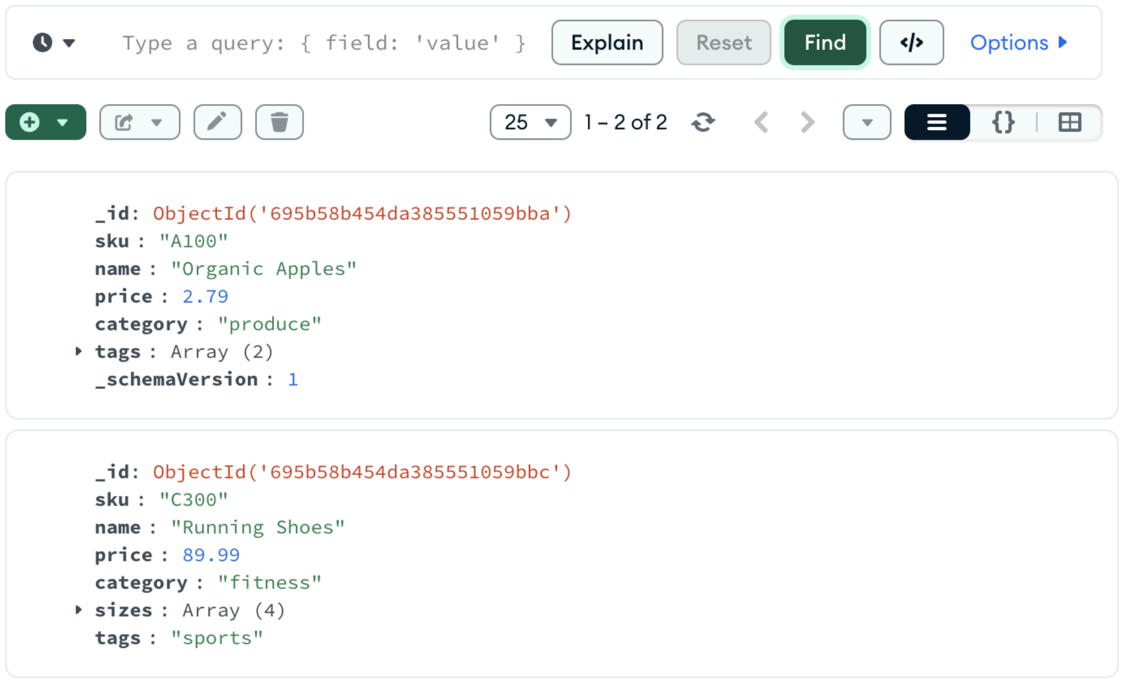
```
_id: ObjectId('695b58b454da385551059bbc')
sku : "C300"
name : "Running Shoes"
price : 89.99
category : "fitness"
sizes : Array (4)
tags : "sports"
```

Part F: Task F1, Task F2



```
_id: ObjectId('695b5af054da385551059bbe')
sku : "B200"
name : "Herbal Toothpaste"
price : 4.99
category : "wellness"
tags : Array (2)
```

Document flagged for deletion.



```
_id: ObjectId('695b58b454da385551059bba')
sku : "A100"
name : "Organic Apples"
price : 2.79
category : "produce"
tags : Array (2)
_schemaVersion : 1
```

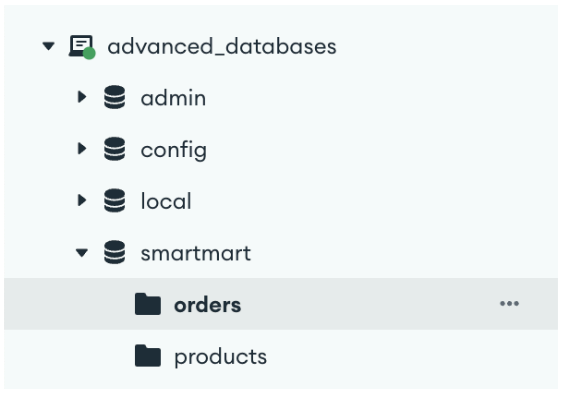
```
_id: ObjectId('695b58b454da385551059bbc')
sku : "C300"
name : "Running Shoes"
price : 89.99
category : "fitness"
sizes : Array (4)
tags : "sports"
```

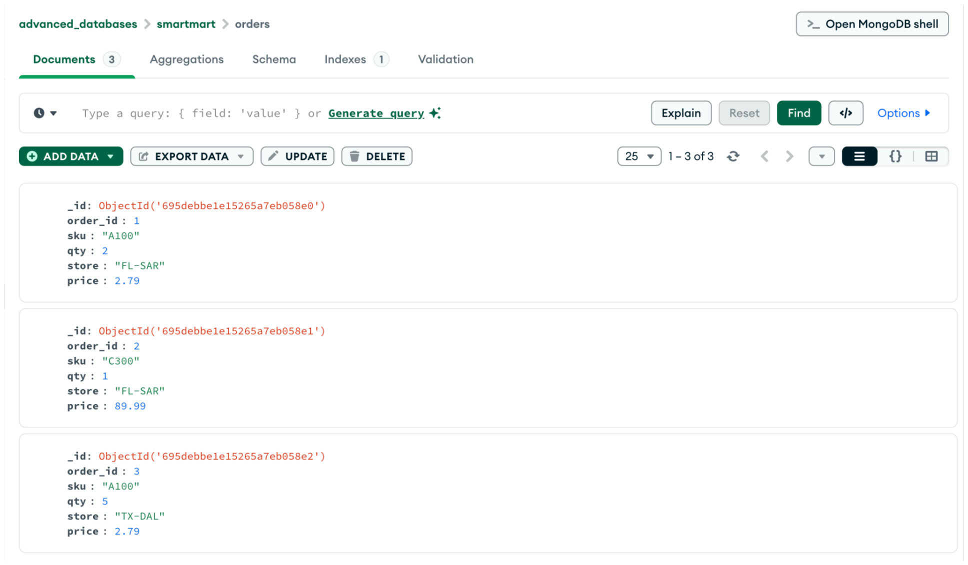
<u>Summary</u>

In the Lab today, we downloaded MongoDB Compass and Server. We then created the database for SmartMart, followed by the Products collection. In Task C, we inserted the 3 items. In Task D, we filtered to find items where category is “wellness” and “produce”. We were unable to find an item that matched {name: 1, price: 1, _id: 0}. In Task E, we modified the price of item “A100”, as well as added a tag to “C300”. Lastly, in Task F, we removed item B200 from the collection.

Module 10 - Banking domain

<u>Part A: Dataset preparation (A1: Create “orders” collection, A2: Insert sample orders)</u>



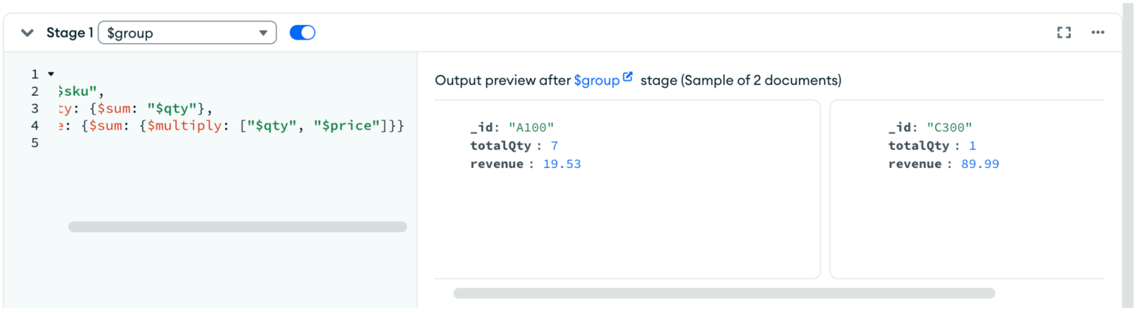


```
_id: ObjectId('695debbe1e15265a7eb058e0')
order_id : 1
sku : "A100"
qty : 2
store : "FL-SAR"
price : 2.79
```

```
_id: ObjectId('695debbe1e15265a7eb058e1')
order_id : 2
sku : "C300"
qty : 1
store : "FL-SAR"
price : 89.99
```

```
_id: ObjectId('695debbe1e15265a7eb058e2')
order_id : 3
sku : "A100"
qty : 5
store : "TX-DAL"
price : 2.79
```

Part B: Aggregation Pipeline (B1: Total sales by SKU, B2: Filter High-Revenue products)

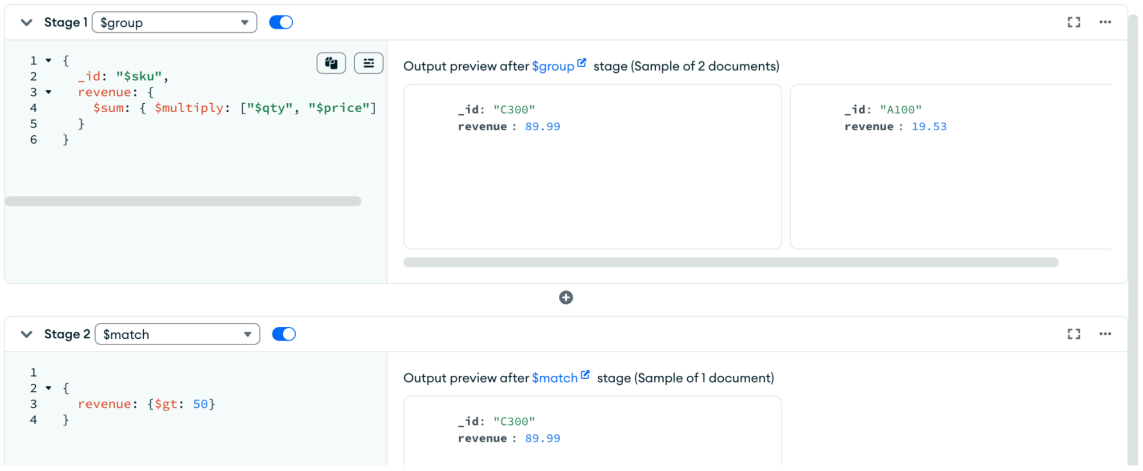


```

}sku",
ty: {$sum: "$qty"},
e: {$sum: {$multiply: ["$qty", "$price"]}}

```

Output preview after $group stage (Sample of 2 documents)

```
_id: "A100"
totalQty : 7
revenue : 19.53
```

```
_id: "C300"
totalQty : 1
revenue : 89.99
```


Stage 1
$group
Output preview after $group stage (Sample of 2 documents)

```
{
  _id: "$sku",
  revenue: {
    $sum: { $multiply: ["$qty", "$price"]
  }
}
```

```
_id: "C300"
revenue : 89.99
```

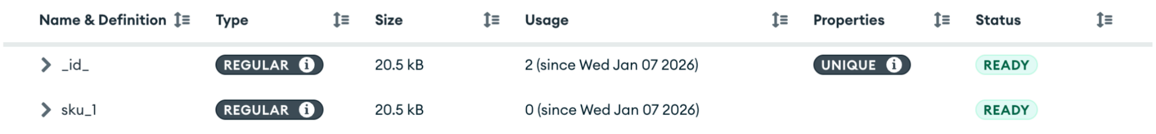
```
_id: "A100"
revenue : 19.53
```

Stage 2
$match
Output preview after $match stage (Sample of 1 document)

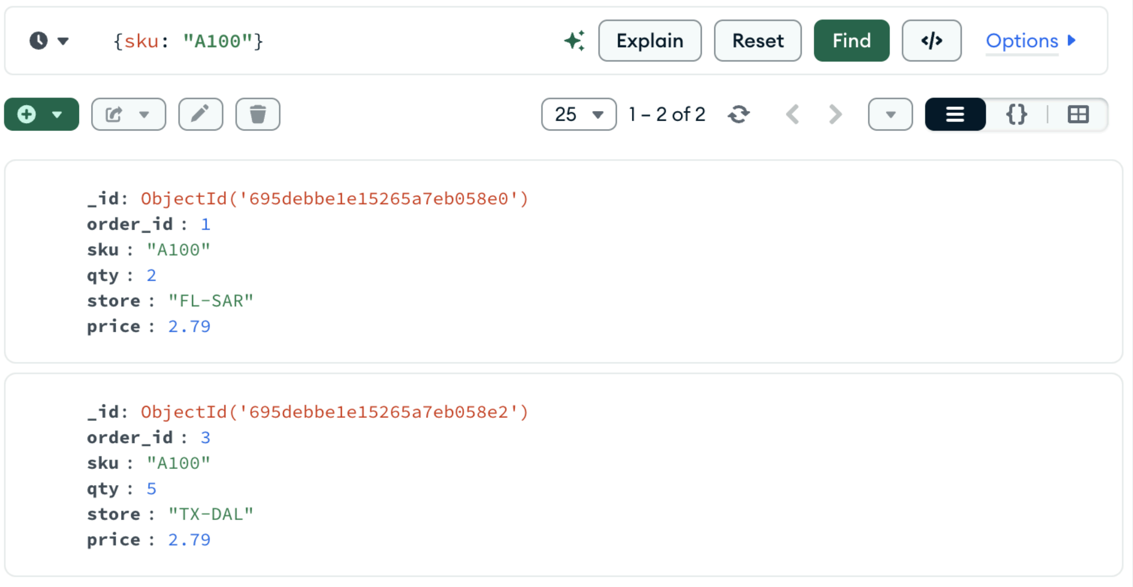
```

{
  revenue: {$gt: 50}
}
```

```
_id: "C300"
revenue : 89.99
```

Part C: Indexing (C1: Create index, C2: Explain query)



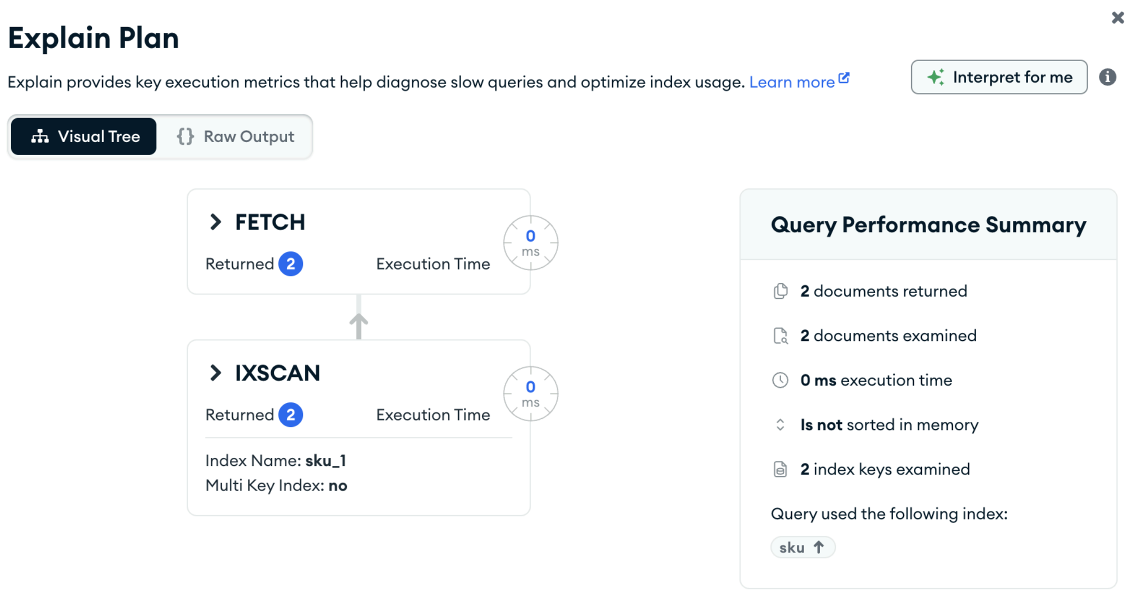
Name & Definition	Type	Size	Usage	Properties	Status
_id_	REGULAR	20.5 kB	2 (since Wed Jan 07 2026)	UNIQUE	READY
sku_1	REGULAR	20.5 kB	0 (since Wed Jan 07 2026)		READY



```
{sku: "A100"}
```

```
_id: ObjectId('695debbe1e15265a7eb058e0')
order_id : 1
sku : "A100"
qty : 2
store : "FL-SAR"
price : 2.79
```

```
_id: ObjectId('695debbe1e15265a7eb058e2')
order_id : 3
sku : "A100"
qty : 5
store : "TX-DAL"
price : 2.79
```



Explain provides key execution metrics that help diagnose slow queries and optimize index usage. Learn more

FETCH
Returned 2 Execution Time 0 ms

IXSCAN
Returned 2 Execution Time 0 ms

Index Name: **sku_1**
Multi Key Index: **no**

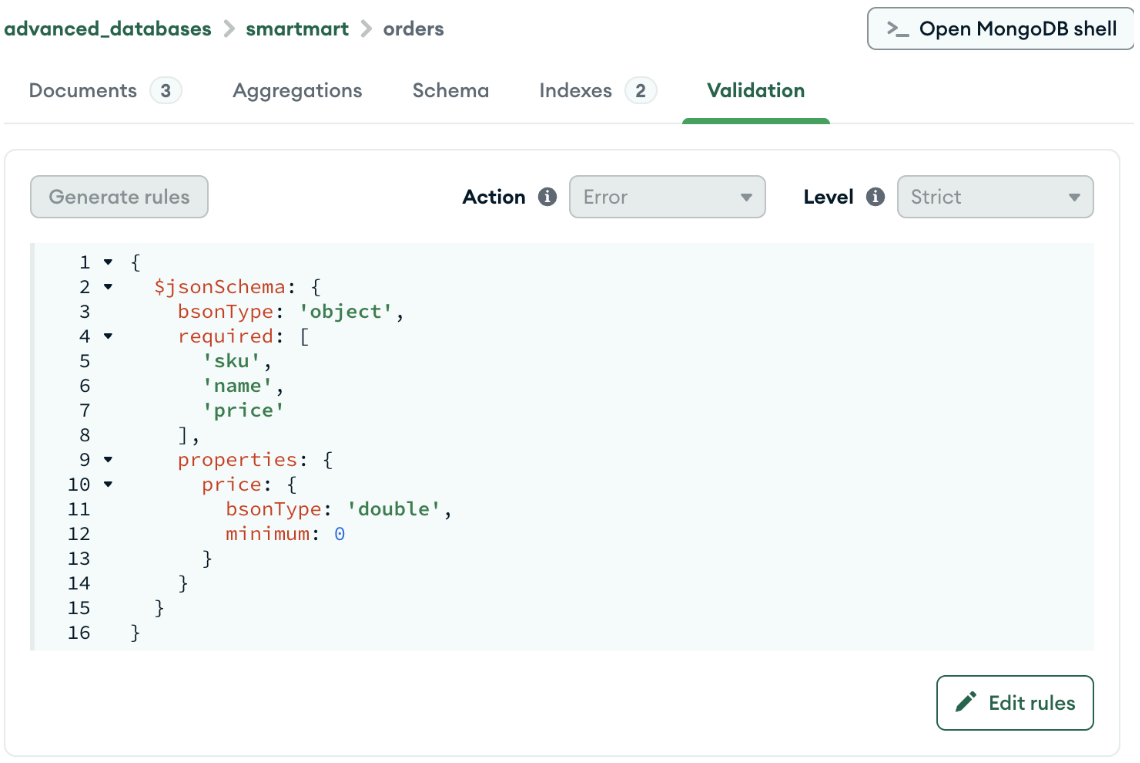
Query Performance Summary

- **2** documents returned
- **2** documents examined
- **0 ms** execution time
- **Is not** sorted in memory
- **2** index keys examined

Query used the following index:

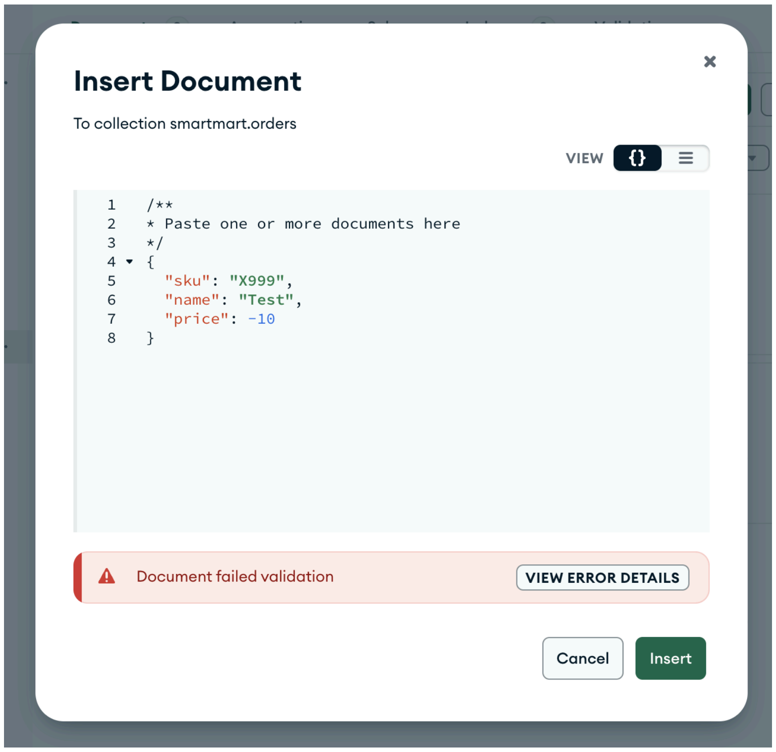
sku ↑

Part D: Schema validation (D1: Apply validation, D2: Test invalid insert)



Action: Error
Level: Strict

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'sku',
      'name',
      'price'
    ],
    properties: {
      price: {
        bsonType: 'double',
        minimum: 0
      }
    }
  }
}
```



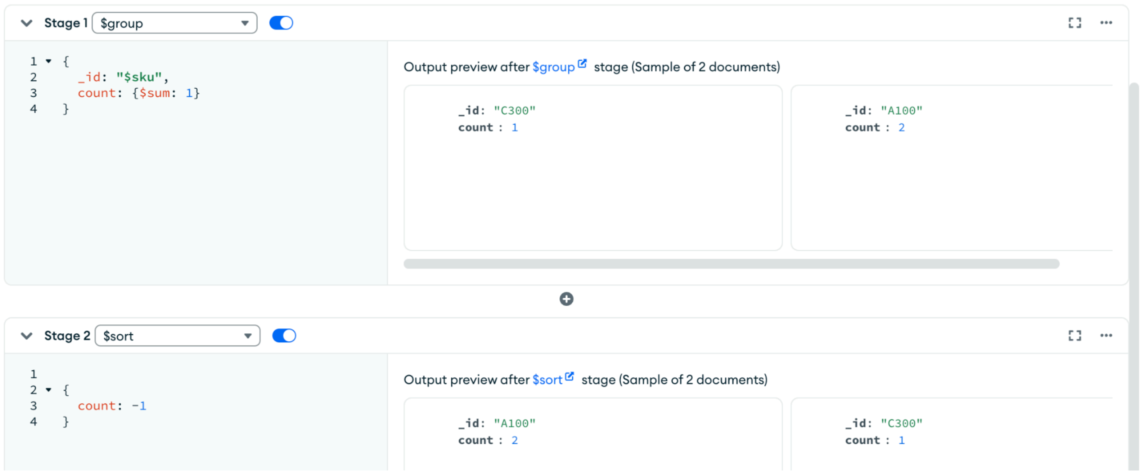
Insert Document

To collection smartmart.orders

```
/**
* Paste one or more documents here
*/
{
  "sku": "X999",
  "name": "Test",
  "price": -10
}
```

Document failed validation

Part E: Long-Tail Analytics (E1: Identify Co-Purchased SKUs)



Stage 1 $group

```
{
    _id: "$sku",
    count: {$sum: 1}
}
```

Output preview after $group stage (Sample of 2 documents)

```
_id: "C300"
count : 1
```

```
_id: "A100"
count : 2
```

Stage 2 $sort

```

{
    count: -1
}
```

Output preview after $sort stage (Sample of 2 documents)

```
_id: "A100"
count : 2
```

```
_id: "C300"
count : 1
```

Part F: MongoDB in Business Architecture

Within the parameters of the 3 requirements specified in the Module 10 slides *(role in AI feature pipeline, integration with ML dashboards, trade-offs vs relational databases)*, our team brainstormed the below applications.

1. **Role in AI Feature Pipelines**

MongoDB is an ideal high-throughput feature store for our fraud detection and credit scoring models with reference to the Risk/Compliance modules as it can aggregate these disparate data points into rich, pre-computed "Customer 360" documents.

Along with the strict transactional integrity of PostgreSQL clusters, this new combined structure allows our ML models to retrieve a single, comprehensive feature vector in milliseconds during inference, significantly reducing latency for real-time fraud scoring compared to querying your partitioned SQL warehouse.

2. **Integration with Dashboards and ML**

For the Admin Dashboard UI and BankStaff interfaces defined in Module 4, MongoDB acts as a flexible operational data layer that bridges our vertically sharded infrastructure.

Using MongoDB Charts etc. allows our team to build real-time dashboards that overlay external economic indicators against our internal TransactionStatus flows, allowing for immediate notification of monitoring spikes in "Failed" or "Pending" states across shards

3. **Trade-offs vs Relational Warehouses**

Finally, while our aforementioned PostgreSQL OLAP cluster is superior for structured, complex join-heavy queries (such as auditing the exact financial lineage between Transactions and TransactionLines), it unfortunately struggles with the agility required for rapidly evolving data types.

The primary trade-off of introducing MongoDB is therefore the *relaxation of strict ACID compliance* for multi-document transactions in favor of write performance and schema flexibility.

Module 11 - Banking domain

Part 1: Analytical, Storage and Infrastructure Technologies for your Team Project

1. ***Analytical Technology: PostgreSQL OLAP Cluster (Citus/PostgresXL)***

For our team's Compliance/Risk module, we designated a *read-only PostgreSQL OLAP cluster* to handle high-compute queries like ViewPotentialFraudTransactions.

This technology would potentially enable our system to run complex joins between the Customers and Transactions tables without locking the primary transactional databases, allowing for real-time fraud detection and long-tail analytics on customer churn.

2. ***Storage Technology: MongoDB (Document Store)***

As covered previously in Module 7 and Module 9, MongoDB was introduced by Prof Bhuvan as the secondary storage layer to handle alternative data and long-tail marketing needs.

While our team decided that PostgreSQL is to remain the system of record for core banking ledgers, we also decided to implement MongoDB to store semi-structured data sources (such as FRED API economic indicators and Kaggle financial datasets that we mentioned in Module 9) to allow our client bank to ingest diverse schema formats (JSON/CSV) rapidly.

3. ***Infrastructure Technology: Kafka CDC (Debezium)***

Our team's database architecture in Module 5 explicitly prioritizes a "Master Data Replication" framework that uses Kafka CDC.

This infrastructure decision was critical given that we decided to allow for streaming updates from the "Corporate" and "Retail" shards to the "Compliance" shard in near real-time to ensure that the fraud detection models always operate on the latest transaction states without direct coupling to the primary shards.

Part 2: Database integration approach

Our team chose to adopt the hybrid sharding strategy taught by Prof Bhuvan in Module 5, so as to bridge the rigid transactional needs of banking with the flexible analytical needs required for our banking domain. We separated our approach into 2 stages below, namely our core and edge integrations.

1. ***Core Integration (PostgreSQL on RDS/Aurora)***

The Corporate, Retail, and Compliance modules will reside on managed PostgreSQL instances.

We will implement the report's Vertical Sharding logic by deploying the Corporate Shard (high-value, low-volume) on a memory-optimized instance to handle complex "High Account Balance" rules.

On the other hand, the Retail Shard (high-volume) will utilize a horizontally scaled cluster to distribute the checking/savings load across the many hash partitions defined in Module 5.

2. ***Edge Integration (MongoDB Atlas + Kafka Connect)***

To integrate the alternate data we introduced in Module 8 without polluting our core schema, we will deploy MongoDB Atlas as a sidecar Operational Data Layer.

Additionally, to stream "Transaction" events into MongoDB documents, we will use Kafka Connect to tap into the RDS transaction logs to enrich them with on-the-fly external FRED/IMF economic data mentioned in Module 8.

This pipeline creates a "Single View" for the Admin Dashboard UI, enabling querying of "churn risk" or "cross-sell opportunities" directly from MongoDB without impacting the performance of the core banking ledger.

Part 3: Approach to improving p95 for a 5-second response time

To ensure a p95 response time well under 5 seconds across the "Retail Customer UI" and "Admin Dashboard," we will target the bottlenecks identified in the Module 5 partitioning diagram using a three-pronged optimization strategy.

1. ***Enforcing Deterministic Routing (Partition Pruning)***

Given that the primary latency killer in sharded systems is nearly always "scatter-gather" queries that query every shard, our team decided to strictly enforce the Horizontal Sharding routing logic defined in Module 5.

By ensuring the application layer resolves the specific retail_sav_2 shard ID before the query is sent, we can therefore guarantee that 95% of customer queries hit a single database instance, eliminating network overhead and ensuring index utilization.

2. ***Materialized Views for Complex Aggregations***

Instead of calculating SUM(TransactionAmount) on every page load, our UI will simply SELECT * from the pre-warmed view to move the computational cost to the background writer process, ensuring the user-facing read is instantaneous and responsive.

3. ***Read/Write Splitting for Compliance Workloads***

To avoid locking rows and spiking our application’s latency when running the complex Compliance and Fraud detection rules defined in our Module 4, our team decided to route all such analytical and "Admin Dashboard" queries to the PostgreSQL Read-Only Replicas or the dedicated OLAP Cluster.

This isolates the "Retail Customer UI" from internal bank operations, ensuring that a massive compliance audit query never degrades the login or transfer speed for a retail user

---

Module 12 - Banking domain

(a) Quality of Data

Our banking domain faces significant data quality risks like inaccuracy, incompleteness, inconsistency, and timeline issues in customer and transaction data. This is exacerbated by high-volume NoSQL injection, schema drift, duplication, and null creep.

(b) Security of Database

Our Strategy: