

INFSCI 2711

Advanced Database Management System

InstaCart Data Warehouse

Sarthak Killedar (SMK178)

Rupali Shyam (RUS37)

Sayantani Bhattacharjee (SAB301)

Dilisha Naidu (DIN10)

Laxmi Ravi (LAR146)

Sanzil Madye (SSM59)

Overview:	3
Assumptions:	3
Description of the data:	3
Relational Model:	4
Data Loading and Pre-processing:	4
STAR Schema Design:	4
Data Analysis:	7
Evaluation:	11
MongoDB :	12
Mapping queries :	12
Creating Index :	12
Creating Look-up collections :	13
Data Analysis :	13
Evaluation:	16
Limitations and Challenges :	16
Neo4j	17
Data Loading and Pre-processing:	17
Creating Indexes	17
Relationship between Nodes	20
Data Analysis	23
Advantages	26
Speed of the queries, navigation through graphs and Data visualization	26
Limitations and Challenges	26
Conclusion	26

Overview:

Data Analysis can help us gain useful insight on the working and performance of a business. This helps such optimize our system to ensure increased profits and better management of the resources. The main aim of this project is to implement a data analysis website for E-commerce data. The website will show important aggregate results related to sale and orders. This can help estimate sales based on Department, Products, Users as well as time. An attempt is also made to implement the analysis using three different database models. One using a relational database and the two using non-relational databases such as MongoDB and Neo4j.

Assumptions:

For the relational model, since the data does not specify the quantity of each product that is ordered, it is assumed to be one. That is, only one item of a given product is added to each order.

Description of the data:

The Instacart dataset is used for the data warehousing and analysis. The InstaCart dataset contain 5 csv file with each of the files containing information about the Aisles, Departments, Products, Orders and Order details. The Aisles file gives us information about the Aisle ID and name associated with each Aisle. Similarly, the Department table contains the Department ID and name. The Orders file contains information about the when the order was placed and who placed the order. It contains the Order ID, User ID of the user who placed the order, the order number, the day of the week the order hour at which the order was placed, and the time since the last order was placed by that customer. Finally, the order details file gives us information about the products ordered. It contains the order ID, the product ID, the cart number and the number of reorders for that product.

Relational Model:

Data Loading and Pre-processing:

As discussed, the data is stored in csv files. This data needs to be cleaned and loaded into appropriate structures to carry out aggregation and analysis on it.

For the relational model, we use python code to read each of the csv files and load it into appropriate relational tables. The code creates a table for each csv file and reads each line of the file one after the other. Each line is split into columns using the comma delimiter. Then each row is inserted into the table.

STAR Schema Design:

For this project, 4 Fact tables have been created – ProductFact, ProductUserFact, OrderUserFact, and ProductDayOfWeekHourFact. MSSQL is used to implement the DDL and DML statements. Each of these dimension tables and fact tables are describes in detail below.

Dimension Tables:

Aisle :

```
CREATE TABLE [dbo].[Aisle](
    [AisleId] [int] NOT NULL,
    [Aisle] [nvarchar](50) NOT NULL,
    CONSTRAINT [PK_Aisles] PRIMARY KEY CLUSTERED
(
    [AisleId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Department:

```
CREATE TABLE [dbo].[Department](
    [DepartmentId] [int] NOT NULL,
    [Department] [nvarchar](50) NOT NULL,
    CONSTRAINT [PK_Department] PRIMARY KEY CLUSTERED
(
    [DepartmentId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Product:

```
CREATE TABLE [dbo].[Product](
    [ProductId] [int] NOT NULL,
    [Name] [nvarchar](max) NOT NULL,
    [AisleId] [int] NOT NULL,
    [DepartmentId] [int] NOT NULL,
    CONSTRAINT [PK_Product] PRIMARY KEY CLUSTERED
(
    [ProductId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Order:

```
CREATE TABLE [dbo].[Order](
    [OrderId] [int] NOT NULL,
    [UserId] [int] NOT NULL,
    [Evaluation] [nvarchar](50) NOT NULL,
    [OrderNumber] [int] NOT NULL,
    [DayOfWeek] [int] NOT NULL,
    [Hour] [int] NOT NULL,
    [SinceLast] [decimal](18, 10) NULL,
    CONSTRAINT [PK_Order] PRIMARY KEY CLUSTERED
(
    [OrderId] ASC,
    [UserId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

OrderDetails:

```
CREATE TABLE [dbo].[OrderDetails](
    [OrderId] [int] NOT NULL,
    [ProductId] [int] NOT NULL,
    [CartOrder] [int] NOT NULL,
    [Reordered] [int] NOT NULL,
    CONSTRAINT [PK_OrderDetails] PRIMARY KEY CLUSTERED
(
    [OrderId] ASC,
    [ProductId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Next, the fact tables and their DDL statements are given below.

ProductStarFact: This fact table aggregates the quantity of each product along with it's associated department and the aisle it belongs to. The quantity is aggregated by joining the

Products and OrderDetails dimension tables and grouped based on Product ID, Department ID and Aisle ID.

```
CREATE TABLE [dbo].[ProductStarFact](
    [ProductId] [int] NOT NULL,
    [DepartmentId] [int] NOT NULL,
    [AisleId] [int] NOT NULL,
    [Quantity] [int] NOT NULL,
    CONSTRAINT [PK_ProductStarFact] PRIMARY KEY CLUSTERED
    ([ProductId] ASC)
    WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

```
INSERT INTO ProductStarFact (ProductId,DepartmentId,AisleId,Quantity)
SELECT [A].[ProductId], [B].[DepartmentId],[B].[AisleId], COUNT(*)
FROM [OrderDetails] AS [A] INNER JOIN [Product] AS [B]
ON [A].[ProductId] = [B].[ProductId]
GROUP BY [A].[ProductId],[B].[DepartmentId],[B].[AisleId]
```

ProductUserStarFact: This fact table aggregates the quantity of products ordered by each user. This table makes use of the order details and the products dimensions and grouped for each Product ID and User ID pair.

```
CREATE TABLE [dbo].[ProductUserStarFact](
    [ProductId] [int] NOT NULL,
    [UserId] [int] NOT NULL,
    [Quantity] [int] NOT NULL,
    CONSTRAINT [PK_ProductUserStarFact] PRIMARY KEY CLUSTERED
    ( [ProductId] ASC,
    [UserId] ASC
    )WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

```
INSERT INTO ProductUserStarFact
(ProductId,UserId,Quantity)
SELECT [A].[ProductId], [B].[UserId], COUNT(*)
FROM [OrderDetails] AS [A] INNER JOIN [Order] AS [B]
ON [A].[OrderId] = [B].[OrderId]
GROUP BY [A].[ProductId],[B].[UserId]
```

OrderUserStarFact: This fact table aggregates the number of orders per user. It makes use of the Orders and OrderDetails dimension tables.

```
CREATE TABLE [dbo].[OrderUserStarFact](
    [OrderId] [int] NOT NULL,
    [UserId] [int] NOT NULL,
```

```

[Quantity] [int] NOT NULL,
CONSTRAINT [PK_OrderUserStarFact] PRIMARY KEY CLUSTERED
(
    [OrderId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]

```

```

INSERT INTO OrderUserStarFact
(OrderId,UserId,Quantity)
SELECT [A].[OrderId], [A].[UserId],COUNT(*)
FROM [Order] AS [A] INNER JOIN [OrderDetails] AS [B]
ON [A].[OrderId] = [B].[OrderId]
GROUP BY [A].[OrderId],[A].[UserId] ORDER BY OrderId

```

ProductDayOfWeekHourStarFact: This fact table aggregates the details of orders placed based on time, that is the day of the week and hour at which a given product is ordered.

```

CREATE TABLE [dbo].[ProductDayOfWeekHourStarFact](
    [ProductId] [int] NOT NULL,
    [DayOfWeek] [int] NOT NULL,
    [Hour] [int] NOT NULL,
    [Quantity] [int] NULL,
CONSTRAINT [PK_ProductDayOfWeekStarFact] PRIMARY KEY CLUSTERED
(
    [ProductId] ASC,
    [DayOfWeek] ASC,
    [Hour] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]

```

```

INSERT INTO ProductDayOfWeekHourStarFact
SELECT [B].[ProductId], [A].[DayOfWeek], [A].[Hour], COUNT(*) FROM [Order] AS [A]
INNER JOIN [OrderDetails] AS [B]
ON [A].[OrderId] = [B].[OrderId]
GROUP BY [B].[ProductId], [A].[DayOfWeek], [A].[Hour]

```

We can convert the above queries to a stored procedure which can handle modified data as well as new data. Hence for each day, the queries will be updated based on the latest values in the dimension tables.

Data Analysis:

In the Relational model, data analysis is performed on understand the pattern of orders and sales based on the product, user and time.

1. Total number of orders per department displayed from highest to lowest

```

SELECT Department, SUM([A].[Quantity]) AS NumberOfOrders

```

```

FROM ProductStarFact AS [A]
INNER JOIN [Department] AS [B]
ON [A].[DepartmentId] = [B].[DepartmentId]
GROUP BY [B].[Department]
ORDER BY NumberOfOrders DESC;

```

Department by Number of Orders

Department	Number of Orders
produce	409087
dairy eggs	217051
snacks	118862
beverages	114046
frozen	100426
pantry	81242
bakery	48394
canned goods	46799

2. Top 5 users that have ordered the most products.

This would give an idea of the top 5 customers and customize promotions and deals for valuable customers.

```

SELECT TOP(5) NEWID() AS [Id], [A].[UserId],
SUM([A].[Quantity]) AS [Quantity]
FROM OrderUserStarFact [A]
GROUP BY [A].[UserId]
ORDER BY [Quantity] DESC;

```

Top 5 Users by Order Quantity

User Id	Quantity
197541	80
149753	80
63458	77
83993	76
189951	76

3. The quantity of products in each Aisle.

This would help in foreseeing the inventory status and adding in more products before the products run out of stock.


```

SELECT [B].[Aisle], SUM([A].[Quantity]) AS [Quantity]
FROM [ProductStarFact] AS [A]
INNER JOIN [Aisle] AS [B]
ON [A].[AisleId] = [B].[AisleId]
GROUP BY [B].[Aisle]
ORDER BY [Quantity]

```

Aisle by Number of Orders

Aisle	Number of Orders
beauty	287
frozen juice	294
baby accessories	306
baby bath body care	328
kitchen supplies	448
specialty wines champagnes	461
ice cream toppings	504
shave needs	532

4. Maximum orders by day of week

This can help analyse the busiest days and provide essential hardware support accommodate peak demand traffic.

For this query, we first create a new temporary dimensional table to store the day of week and maximum quantity from the ProductDayOfWeekHourStartFact table by grouping by day of week. This dimension table is joined again with the ProductDayOfWeekHourStartFact table to retrieve the product ID as well and create a new ProductDayOfWeekFact table from which the final query is executed.

```

create table #Temp
(
    [DayOfWeek] INT,
    [Quantity] INT
)

```

```

INSERT INTO #Temp
SELECT [DayOfWeek], MAX(TotalQuantity) AS [MaxQuantity] FROM (SELECT [DayOfWeek],
ProductId, sum(Quantity) AS TotalQuantity
FROM ProductDayOfWeekHourStarFact
GROUP BY [DayOfWeek], ProductId) AS [X] GROUP BY [X].[DayOfWeek]

```

```

SELECT [X].[DayOfWeek], [X].ProductId, [X].[Quantity] FROM (SELECT [A].[DayOfWeek],
[A].ProductId, SUM([A].[Quantity]) AS [Quantity]
FROM ProductDayOfWeekHourStarFact AS [A] GROUP BY [A].[DayOfWeek], [A].[ProductId]) AS [X]
INNER JOIN #Temp AS [A]
ON [X].[DayOfWeek] = [A].[DayOfWeek] AND [X].[Quantity] = [A].[Quantity]

```

```

SELECT CASE [A].[DayOfWeek]
    WHEN 0 THEN 'Sunday'
    WHEN 1 THEN 'Monday'
    WHEN 2 THEN 'Tuesday'
    WHEN 3 THEN 'Wednesday'
    WHEN 4 THEN 'Thursday'
    WHEN 5 THEN 'Friday'
    WHEN 6 THEN 'Saturday' END AS [DayOfWeek], [B].[Name], [A].[TotalQuantity]
FROM ProductDayOfWeekFact AS [A] INNER JOIN [Product] AS [B]
ON [A].[ProductId] = [B].[ProductId]

```

Products By Day of Week

Day of Week	Product Name	Quantity
Sunday	Banana	4705
Monday	Banana	2963
Tuesday	Banana	2121
Wednesday	Banana	2007
Thursday	Banana	1950
Friday	Banana	2291
Saturday	Banana	2689

5. Sales (number of products ordered) depending on hour of day

```

SELECT [TimeOfDay], SUM(Quantity) AS [Quantity]
FROM (SELECT CASE WHEN [Hour] >= 0 AND [Hour] <=3 THEN '00:00 – 03:00'
    WHEN [Hour] >= 4 AND [Hour] <= 6 THEN '03:00 – 06:00'
    WHEN [Hour] >= 7 AND [Hour] <=9 THEN '06:00 – 09:00'
    WHEN [Hour] >= 10 AND [Hour] <=12 THEN '09:00 – 12:00'
    WHEN [Hour] >= 13 AND [Hour] <=15 THEN '12:00 – 15:00'
    WHEN [Hour] >= 16 AND [Hour] <=18 THEN '15:00 – 18:00'
    WHEN [Hour] >= 19 AND [Hour] <=21 THEN '18:00 – 21:00'
    WHEN [Hour] >= 22 AND [Hour] <=24 THEN '21:00 – 24:00'
    END AS [TimeOfDay],

```

SUM(Quantity) AS [Quantity]
 FROM ProductDayOfWeekHourStarFact GROUP BY [Hour]) AS [A]
 GROUP BY [TimeOfDay]

Quantity By Time Of Day

Time of Day	Quantity
00:00 – 03:00	20373
03:00 – 06:00	18125
06:00 – 09:00	197544
09:00 – 12:00	336350
12:00 – 15:00	350330
15:00 – 18:00	283703
18:00 – 21:00	133908
21:00 – 24:00	44284

Evaluation:

<i>No. of Execution</i>	<i>Department by number of orders</i>	<i>Quantity by time of day</i>	<i>Max order by User</i>	<i>Products by day of week</i>	<i>Quantity by Aisle</i>
1	21.28	2479.59	87.84	11.53	36.65
2	14.78	377.48	76.68	08	14.96
3	32.07	308.03	62.47	091	17.96
4	32.35	237.59	62.9	0.8	16.26
5	35.49	262.1	68.38	14.1	14.1
Mean Value	27.19ms	732.89ms	71.65ms	2.9ms	19.98ms

MongoDB :

As we know MongoDB is a noSQL database, following document structure. It's supports a flat structure. Instacart dataset used in this project is highly relational, so we face a few challenges for integrating it in mongodb. Just like we created fact and dimensions table in relational model for ease of access, in mongodb we decided to reduced the number of tables accessed. We mapped aisle id and department id into products to avoid joins on these tables. So we need to access just three collections to work on the query.

Mapping queries :

Product - department mapping :

```
db.products.find().forEach(function (doc1) {  
    var doc2 = db.departments.findOne({ department_id: doc1.department_id }, { department:  
        1 });  
    if (doc2 != null) {  
        doc1.department_id = doc2.department;  
        db.products.save(doc1);  
    }  
});
```

Products - Aisle mapping :

```
db.products.find().forEach(function (doc1) {  
    var doc2 = db.aisles.findOne({ aisle_id: doc1.aisle_id }, { aisle: 1 });  
    if (doc2 != null) {  
        doc1.aisle_id = doc2.aisle;  
        db.products.save(doc1);  
    }  
});
```

Creating Index :

1. Index on order_id in orders and order_product collection to speed up the lookup and group by queries.

```
db.orders.createIndex({ order_id : 1});  
db.order_product.createIndex({ order_id : 1});
```

2. Index on product_id in products and order_product collection to speed up the lookup and group by queries.

```
db.products.createIndex({ product_id : 1});  
db.order_product.createIndex({ product_id : 1});
```

Creating Look-up collections :

1. Look up for orders and order_product collection(user_id, order_id and product_id)

```
db.order_product.aggregate([
  { $lookup: {
    from: "orders",
    foreignField: "order_id",
    localField: "order_id",
    as: "data"
  }},
  { $unwind : "$data"},
  { $project : { _id : 0, user_id : "$data.user_id", order_id : 1, product_id : 1}},
  { $out: "user_order_product" }
]);
```

2. Look up for order_products and products collection (order_id, product_id, product_name, department_id)

```
db.order_product.aggregate([
  { $lookup: {
    from: "products",
    foreignField: "product_id",
    localField: "product_id",
    as: "data"
  }},
  { $unwind : "$data"},
  { $project : { _id : 0, order_id : 1, product_id : 1, product_name :
    "$data.product_name", department : "$data.department_id"}},
  { $out: "order_product_detail" }
]);
```

Data Analysis :

Data analysis is performed to understand the patterns in orders and derive insights regarding products based on orders placed and time when they were placed. We also compare the execution time with other models.

Queries :

1. Top 20 products ordered :

```
db.order_product_detail.aggregate([
  { $group : { _id: "$product_name", product_count: { $sum: 1 } }},
  { $sort : { product_count : -1 }},
  { $limit : 20 }
]);
```

Top 20 Products

Product Name	Quantity
Banana	18726
Bag of Organic Bananas	15480
Organic Strawberries	10894
Organic Baby Spinach	9784
Large Lemon	8135
Organic Avocado	7409
Organic Hass Avocado	7293
Strawberries	6494
Limes	6033
Organic Raspberries	5546
Organic Blueberries	4966
Organic Whole Milk	4908
Organic Cucumber	4613
Organic Zucchini	4589

2. Top 20 aisles with maximum number of products :

```
db.products.aggregate([
  {$group : {_id : "$aisle_id", product_count : {$sum : 1}}},
  {$sort : {product_count : -1}},
  {$limit : 20}
]);
```

Top 20 product by aisle

Aisle	Number of Orders
missing	1258
candy chocolate	1246
ice cream ice	1091
vitamins supplements	1038
yogurt	1026
chips pretzels	989
tea	894
packaged cheese	891
frozen meals	880
cookies cakes	874
energy granola bars	832
hair care	816
spices seasonings	797
juice nectars	792

3. Count of orders by hour of the day :

```
db.orders.aggregate([
  {$group : {_id : "$order_hour_of_day", order_count : {$sum : 1}}},
  {$sort : {order_count : -1}},
  {$limit : 20}]);
```

Orders by Hour Of Day

Time of Day	Quantity
10	288418
11	284728
15	283639
14	283042
13	277999
12	272841
16	272553
9	257812
17	228795
18	182912
8	178201
19	140569
20	104292
7	91868

4. Users who ordered maximum products :

```
db.user_order_product.aggregate([
  {$group : {_id : "$user_id", product_count : {$sum : 1}}},
  {$sort : {product_count : -1}},
  {$limit : 5}
]);
```

Top 5 Users ordering maximum products

User Id	Total orders
149753	80
197541	80
63458	77
189951	76
83993	76

5. Order by Product

```
db.order_product_detail.aggregate([
  {$group : {_id:"$department",product_count: {$sum: 1}}},
  {$sort : {product_count : -1}}
]);
```

Order by Department

Department id	Total orders
produce	409087
dairy eggs	217051
snacks	118862
beverages	114046
frozen	100426
pantry	81242
bakery	48394
canned goods	46799
deli	44291
dry goods pasta	38713
household	35986
meat seafood	30307
breakfast	29500
personal care	21570

Evaluation:

<i>No.of Executions</i>	<i>Department by number of orders</i>	<i>Quantity by Hour of day</i>	<i>Top 5 Users</i>	<i>Top 20 Products</i>	<i>Top 20 Product by Aisle</i>
1	1.10	2.78	1.36	2.15	0.108
2	1.10	2.88	1.19	1.77	0.050
3	1.10	2.72	1.13	1.64	0.052
4	1.14	2.68	1.11	1.60	0.051
5	1.06	2.74	1.40	1.69	0.049
Mean Value	1.1 sec	2.76 sec	1.2 sec	1.7 sec	0.062 sec

Limitations and Challenges :

Since mongodb supports document/flat structure for the database, converting this highly relational dataset and mapping the data was challenging. Initially our plan was to map all collections into one huge collection which would have saved us the effort for using joins/lookup. Due to high computational requirements on such a huge dataset (Orders collection 3.4million documents, Order_Product 1 million documents). Mapping these two collections wasn't possible on our local systems. To overcome this limitation, we created

indexes on the collections we couldn't map i.e. orders, order_product, products on fields order_id and product_id and created corresponding lookup collections *user_order_product* and *order_product_detail* which made our aggregation queries a lot easier and faster.

Another challenge we faced was in terms of query execution time. The queries which required us to perform a join and group by on different collections and show the results in array form, the performance of the query was very poor.

Neo4j

Neo4j is a NoSQL graph database management system. In Neo4j, data is stored in the form of an edges, nodes and attributes and it can be manipulated using Cypher Query Language (CQL).

Data Loading and Pre-processing:

Instacart data is downloaded from Kaggle and data cleaning is done according to Neo4j. The Load csv query is used to import the csv files but Neo4j cannot handle ' " ' " ' so we replaced it with space in the orders table where some of the products have double quotes (e.g. " Easy Grab 8\"x8\" Glass Bakeware "). Also, to import into Neo4j db the downloaded files are added in the data path of the DB (instacart) inside the import folder from where Neo4j expects the data. CQLs LOAD queries are used to import the data from csv files and translate it into nodes and relationships. Below are the load queries to import the csv files,

Loading and creating nodes and relationships:

Creating Indexes

It is very important to create indexes so that loading and querying data becomes lot more faster

```
CREATE INDEX ON :Department(name);
CREATE INDEX ON :Department(id);
CREATE INDEX ON :Aisle(id);
CREATE INDEX ON :Aisle(name);
CREATE INDEX ON :Product(name);
CREATE INDEX ON :Product(id);
CREATE INDEX ON :Order(id);
CREATE INDEX ON :User(id);
```

:schema

```
$ :schema

Indexes
  ON :Aisle(id) ONLINE
  ON :Aisle(name) ONLINE
  ON :Department(id) ONLINE
  ON :Department(name) ONLINE
  ON :Order(id) ONLINE
  ON :Product(id) ONLINE
  ON :Product(name) ONLINE
  ON :User(id) ONLINE

No constraints
```

Loading:

Load Departments

```
LOAD CSV WITH HEADERS FROM "file:///departments.csv" as line
CREATE (:Department {id: line.department_id, name: line.department})
```

```
$ LOAD CSV WITH HEADERS FROM "file:///departments.csv" as...
```

Table

Added 21 labels, created 21 nodes, set 42 properties, completed after 17130 ms.

Code

Load Aisles

```
LOAD CSV WITH HEADERS FROM "file:///aisles.csv" as line
CREATE (:Aisle {id: line.aisle_id, name: line.aisle})
```

```
$ LOAD CSV WITH HEADERS FROM "file:///aisles.csv" as...
```

Table

Added 134 labels, created 134 nodes, set 268 properties, completed after 271 ms.

Code

Load Products

```
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///products.csv" AS line
MATCH (a:Aisle {id: line.aisle_id}), (d:Department {id: line.department_id})
```

MERGE (a)-[:ON]-(p:Product {id: line.product_id, name: line.product_name})-[:IN]->(d)

\$ USING PERIODIC COMMIT 10000 LOAD CSV WITH HEADERS ...

↗

↶

^

↺

×

Table

Added 49688 labels, created 49688 nodes, set 99376 properties, created 99376 relationships, completed after 66959 ms.

Code

Load Orders

USING PERIODIC COMMIT 10000

LOAD CSV WITH HEADERS FROM "file:///orders.csv" as line

MERGE (u:User {id: line.user_id})

MERGE (u)-[:ORDERED]->(o:Order {

id: line.order_id,

orderNumber: toInteger(line.order_number),

dayOfWeek: toInteger(line.order_dow),

hourOfDay: toInteger(line.order_hour_of_day)

});

\$ USING PERIODIC COMMIT 10000 LOAD CSV WITH HEADERS FROM...

↗

↶

^

↺

×

Table

Added 225087 labels, created 225087 nodes, set 862044 properties, created 212319 relationships, completed after 299947 ms.

Code

\$ USING PERIODIC COMMIT 10000 LOAD CSV WITH HEADERS FR...

↗

↶

^

↺

×

Table

Added 250483 labels, created 250483 nodes, set 959251 properties, created 236256 relationships, completed after 50775 ms.

Code

Load Order Items

USING PERIODIC COMMIT 10000

LOAD CSV WITH HEADERS FROM "file:///order_products__train.csv" as line

MATCH (o:Order {id: line.order_id}), (p:Product {id: line.product_id})

MERGE (p)-[:IN_ORDER {addToCartOrder: line.add_to_cart_order,

reordered: line.reordered}}->(o)

\$ USING PERIODIC COMMIT 10000 LOAD CSV WITH HEADERS FR...

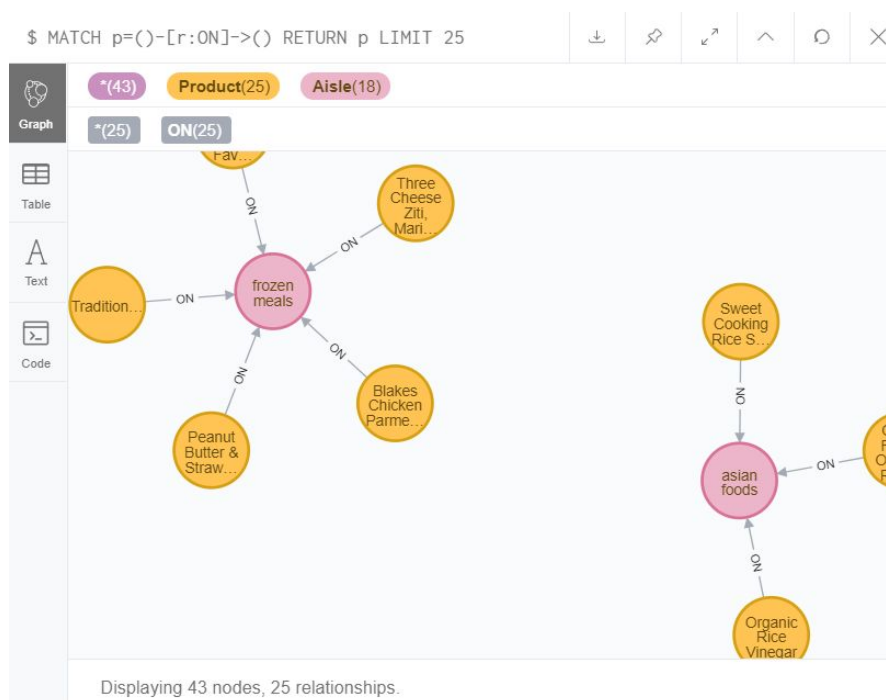
Table

Code

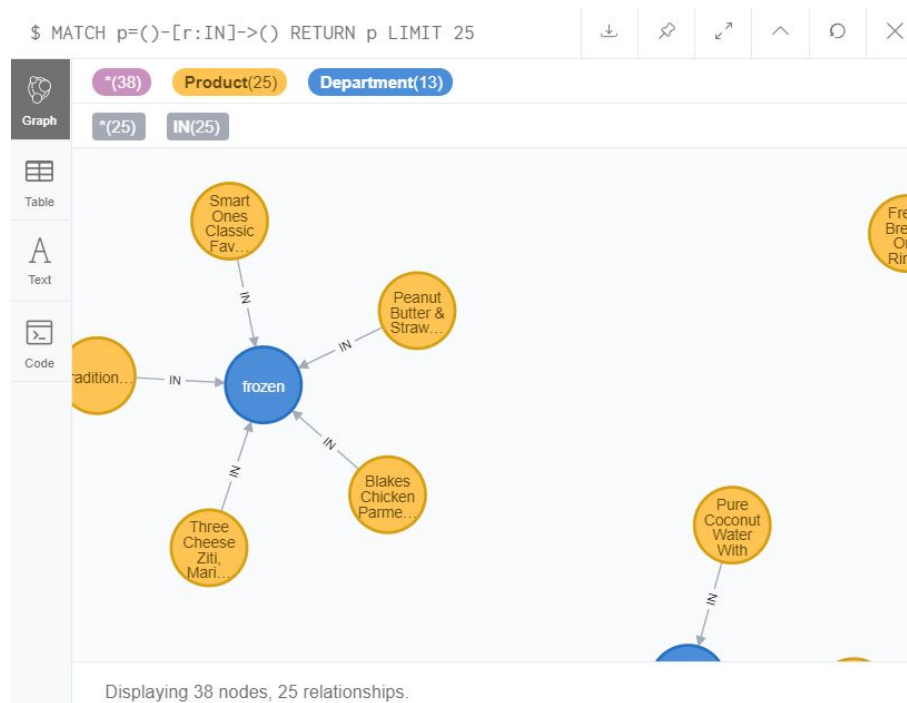
Set 318100 properties, created 159050 relationships, completed after 158174 ms.

Relationship between Nodes

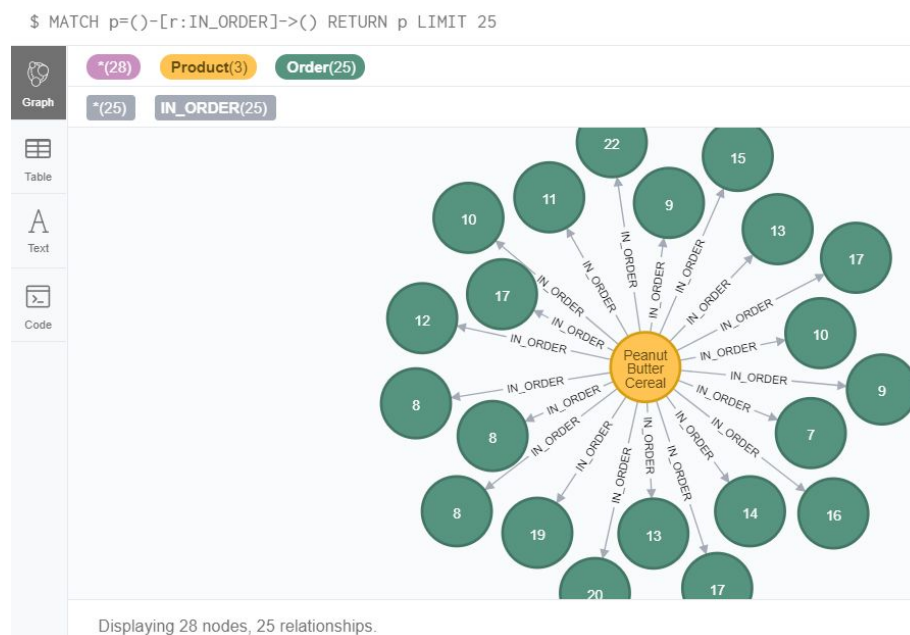
ON relation - Product ON Aisle



IN relation - Product IN Department

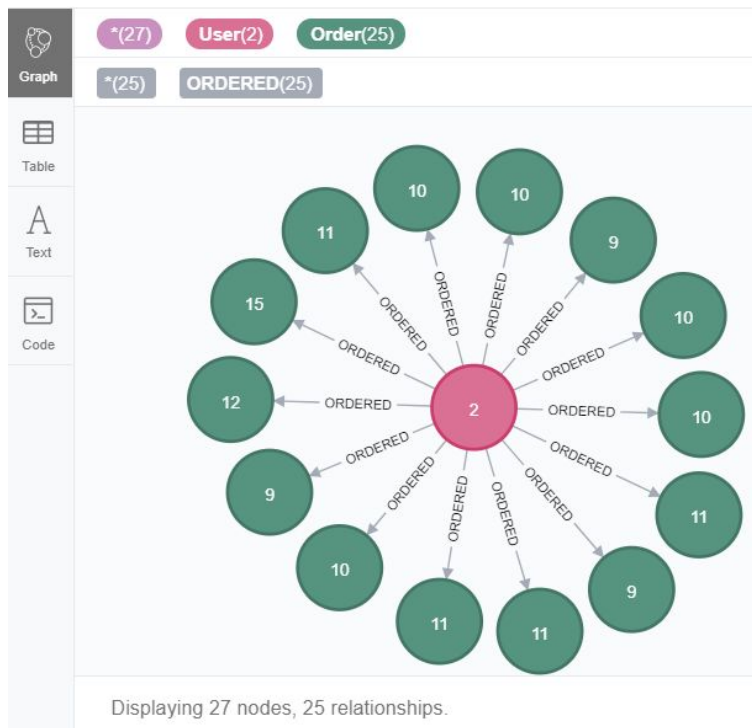


IN-ORDER - Products IN-ORDER OrderId

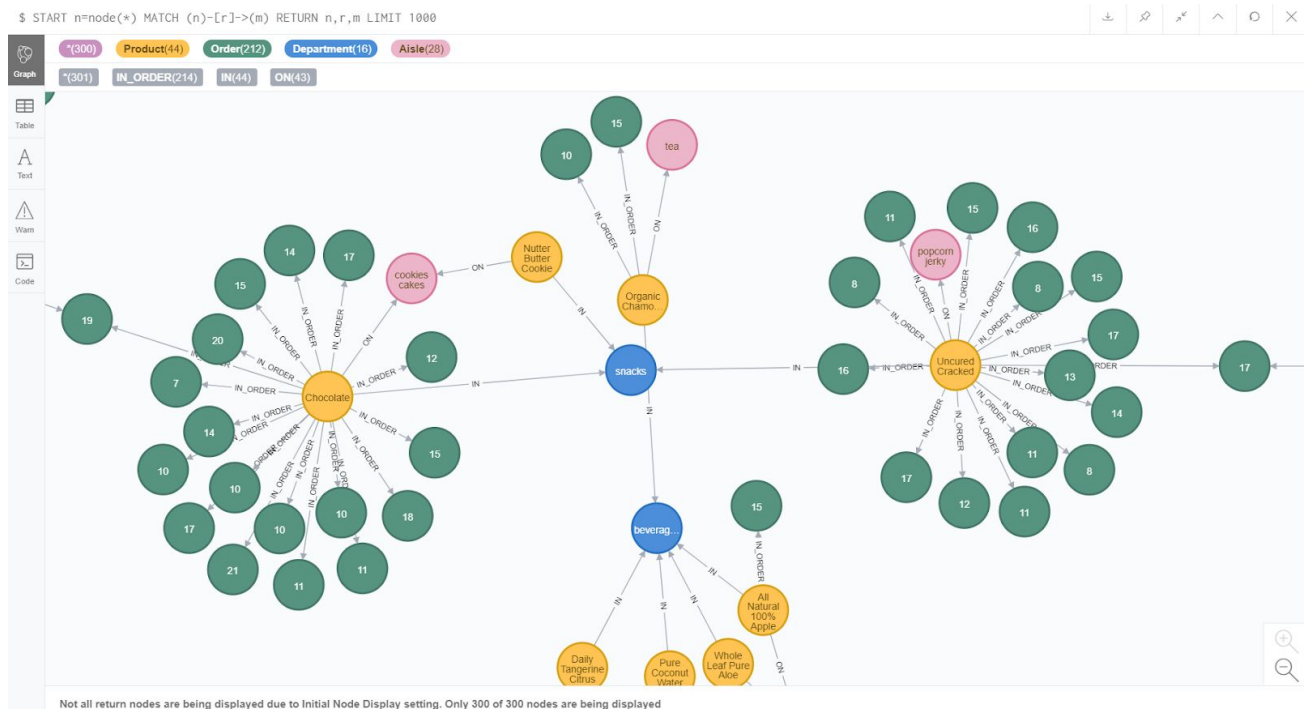


ORDERED - UserId ORDERED OrderId

```
$ MATCH p=()-[r:ORDERED]->() RETURN p LIMIT 25
```



```
START n=node(*) MATCH (n)-[r]->(m) RETURN n,r,m LIMIT 1000;
```

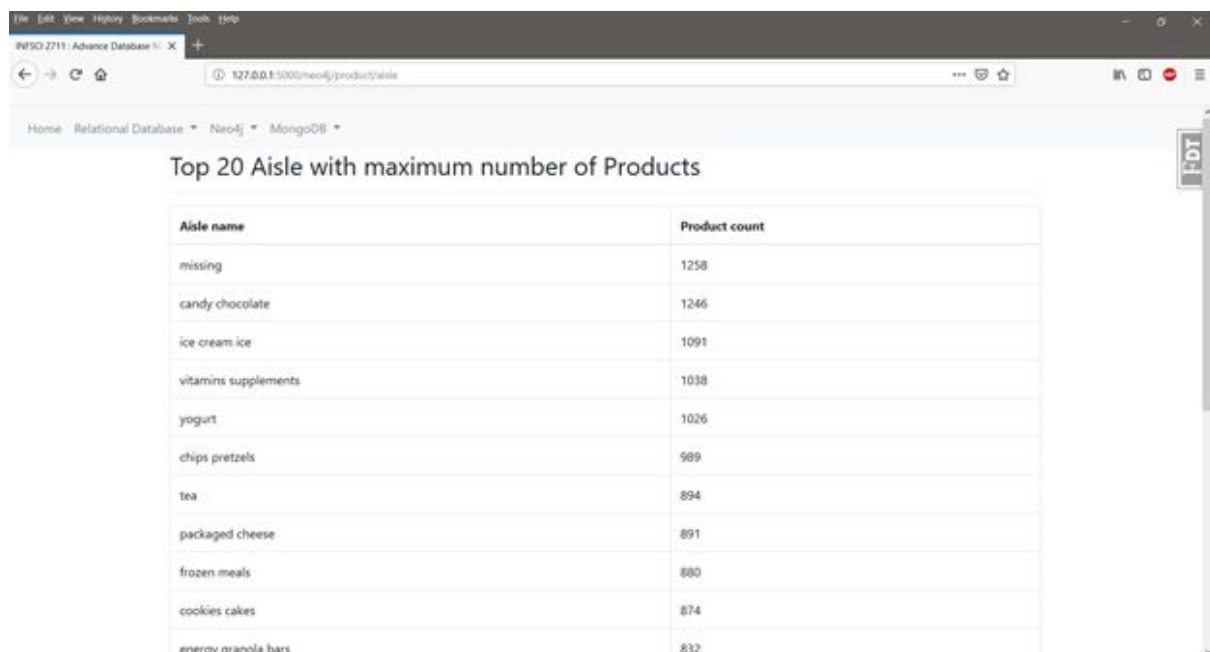


Data Analysis

Data analysis is done to understand relationship between nodes to query different results while comparing the execution time with other models.

Top 20 Aisles with maximum number of products

```
MATCH (p:Product)-[r:ON]->(a:Aisle)
RETURN a.name as Aisle, count(r) AS products_on_aisle
ORDER BY products_on_aisle DESC LIMIT 20
```

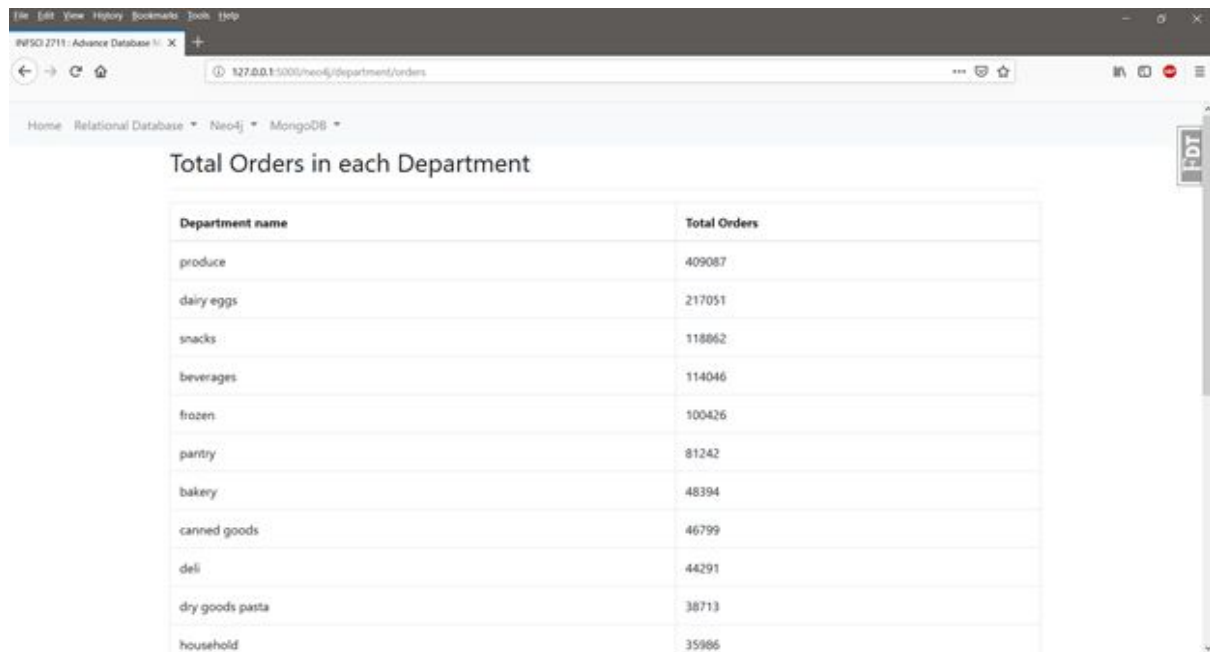
A screenshot of a web browser window displaying a query result. The browser's address bar shows a URL starting with '127.0.0.1:5000/neo4j/product/aisle'. The page title is 'Top 20 Aisle with maximum number of Products'. Below the title is a table with two columns: 'Aisle name' and 'Product count'. The table lists 20 aisles, with 'missing' having the highest count at 1258, followed by 'candy chocolate' at 1246, and 'energy granola bars' at the bottom with 832 products.

Aisle name	Product count
missing	1258
candy chocolate	1246
ice cream ice	1091
vitamins supplements	1038
yogurt	1026
chips pretzels	989
tea	894
packaged cheese	891
frozen meals	880
cookies cakes	874
energy granola bars	832

Started streaming 20 records in less than 1 ms and completed after 48 ms.

2. Total number of orders in each department.

```
MATCH (p:Product)-[r1:IN]->(d:Department),(p)-[r2:IN_ORDER]->(c:Order)
RETURN d.name as department_name,count(r2) AS Number_of_Orders
ORDER BY Number_of_Orders DESC
```



The screenshot shows the Neo4j Desktop interface with a query result titled "Total Orders in each Department". The result is a table with two columns: "Department name" and "Total Orders". The data is as follows:

Department name	Total Orders
produce	409087
dairy eggs	217051
snacks	118862
beverages	114046
frozen	100426
pantry	81242
bakery	48394
canned goods	46799
deli	44291
dry goods pasta	38713
household	35986

Started streaming 21 records in less than 1 ms and completed after 1496 ms.

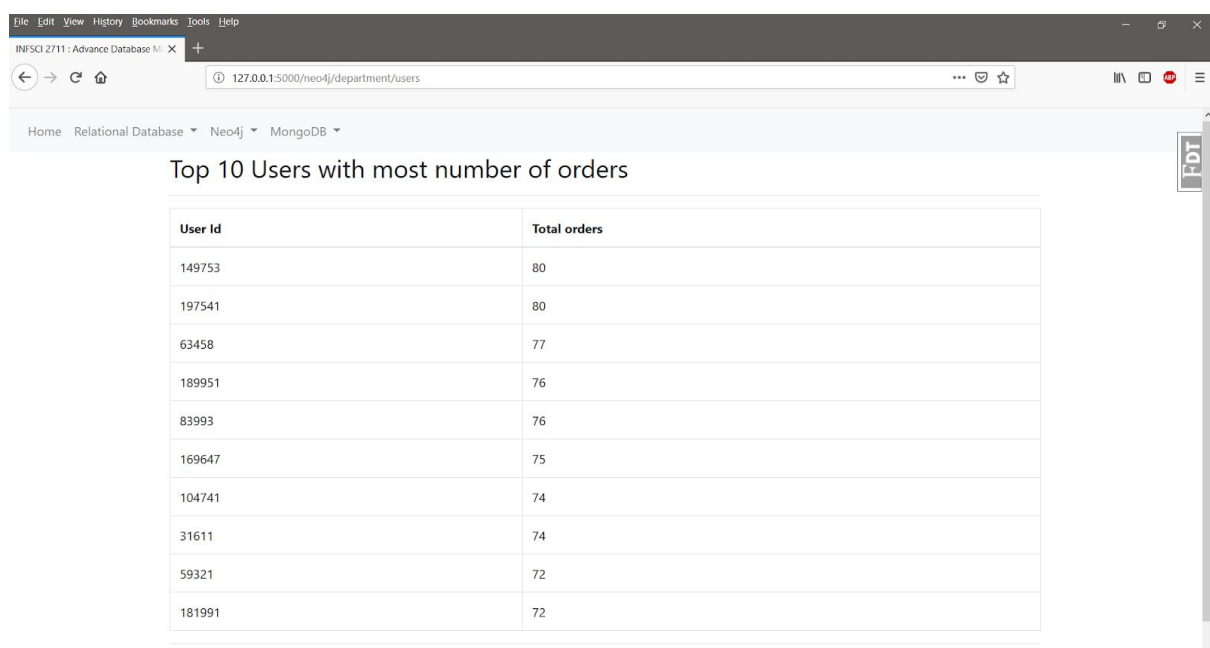
3. Users who ordered most products

MATCH

(u:User)-[r1:ORDERED]->(o:Order),(p:Product)-[r2:IN_ORDER]->(o:Order)

RETURN u.id as User_id,count(r2) AS Quantity

ORDER BY Quantity DESC LIMIT 10"



The screenshot shows the Neo4j Desktop interface with a query result titled "Top 10 Users with most number of orders". The result is a table with two columns: "User Id" and "Total orders". The data is as follows:

User Id	Total orders
149753	80
197541	80
63458	77
189951	76
83993	76
169647	75
104741	74
31611	74
59321	72
181991	72

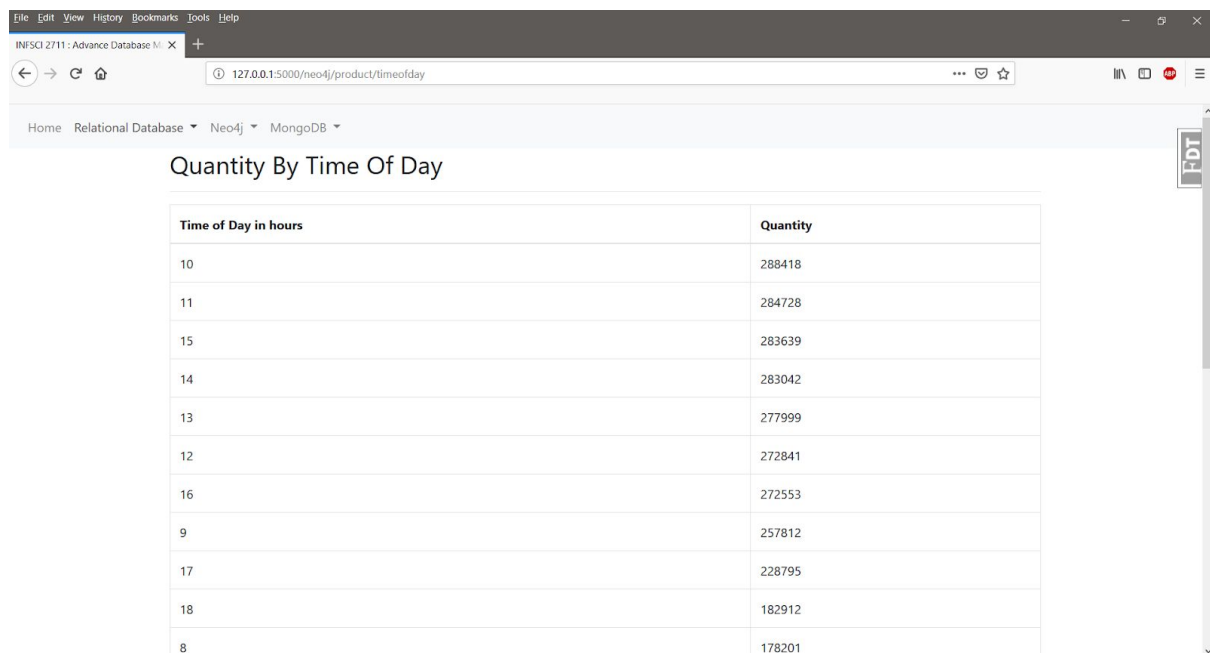
Started streaming 10 records after 1 ms and completed after 5127 ms.

4. Sales by hour

MATCH(o:Order)

RETURN o.hourOfDay as TimeOfDayInHours, COUNT(o.hourOfDay) as salesAtHour

ORDER BY salesAtHour DESC



The screenshot shows a web browser window with a table titled "Quantity By Time Of Day". The table has two columns: "Time of Day in hours" and "Quantity". The data is as follows:

Time of Day in hours	Quantity
10	288418
11	284728
15	283639
14	283042
13	277999
12	272841
16	272553
9	257812
17	228795
18	182912
8	178201

Started streaming 24 records in less than 1 ms and completed after 1374 ms.

5. Products vegetarians buy

MATCH (a:Aisle)-[:ON]-(:Product)-[:IN]->(d:Department)

WHERE (a.name CONTAINS 'meat'

OR a.name CONTAINS 'seafood'

OR d.name CONTAINS 'meat'

OR a.name CONTAINS 'jerky')

AND NOT a.name CONTAINS 'alternatives'

AND NOT a.name CONTAINS 'marinades'

with collect(DISTINCT a.name) AS avoidAisles

MATCH (u:User) WITH u, avoidAisles limit 1000

match (u)-[:ORDERED]->(:Order)-[:IN_ORDER]-(:Product)-[:ON]->(a:Aisle)

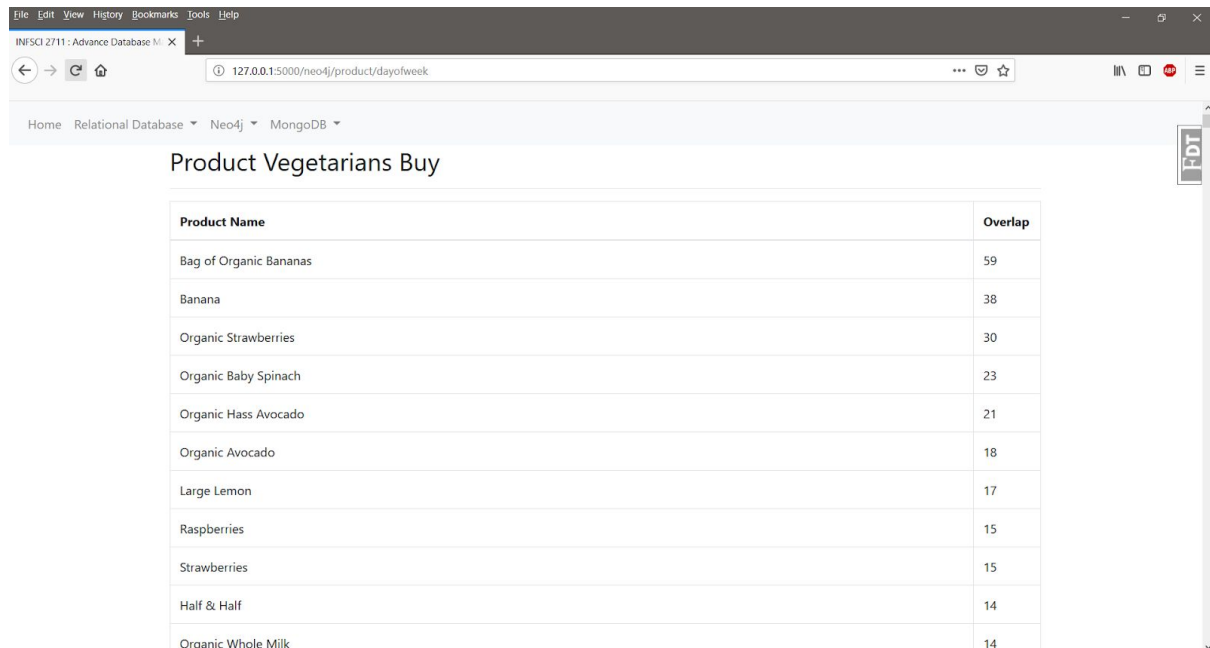
WITH u, collect(DISTINCT a.name) AS aisles, avoidAisles

WHERE none(a IN aisles WHERE a IN avoidAisles)

```

WITH u
MATCH (u)-[:ORDERED]->(:Order)-[:IN_ORDER]-(p:Product)
RETURN p.name, count(*) AS overlap
ORDER BY overlap DESC

```



Product Name	Overlap
Bag of Organic Bananas	59
Banana	38
Organic Strawberries	30
Organic Baby Spinach	23
Organic Hass Avocado	21
Organic Avocado	18
Large Lemon	17
Raspberries	15
Strawberries	15
Half & Half	14
Organic Whole Milk	14

Started streaming 2181 records after 343 ms and completed after 346 ms, displaying first 1000 rows.

Advantages

Speed of the queries, navigation through graphs and Data visualization

Limitations and Challenges

Cypher queries have some limitations - like using regular expressions in place of LIKE in SQL and an alternative for GROUP BY in CQL is COLLECT keyword, which is a highly expensive operation. Also, importing "orders.csv" file which is a large data set slows down the system if the RAM size is insufficient.

Conclusion

From the execution time perspective, Relational model has the best performance for all the analysis queries. Neo4j has a moderate performance followed by MongoDB with the least. The Relational

schema made use of the star schema to develop analytical databases which were efficient in implementing the analysis required.

In terms of implementation, we find that the Relational model had a systematic approach where as the MongoDB models required vital preprocessing steps such as merging tables and creating lookup tables to make the data suitable for query analysis. Neo4j created relations between tables using Cypher Query Language with merge command.