Advanced Data Management Exam

Martin Martuccio (5061529) and Samuele Pellegrini (4951487)

February 2024

Link Presentation:
`https://1drv.ms/v/s!Ahii2763A7PahdUswMxGutViemx1rw?e=fFufRL`

1 Domain and Application Proposal

- **Description of the chosen domain and a relevant application.**

We propose the automotive market as the chosen domain. A related application could be a financial analysis and sales monitoring system for the automotive industry.
The objective of this application would be to collect, manage, and analyze detailed financial data and sales data from sales history. Utilizing a NoSQL database, it could support operations such as inserting new financial data, updating sales information, and executing complex analytical queries to gain strategic insights.
This would enable analysts to conduct comparative analyses, identify market trends, and make decisions within the context of the automotive sector.

2 Details of the Proposed Application

- **Characteristics of the application.**

The application is oriented towards an analytical scenario, with the aim of extracting generic information from a vast historical data plan. The choice of

database is driven by the need to manage a 'read-intensive' workload, known as "batch operation". The system structure emphasizes the priority of reading over writing, implementing data insertion operations limited to simple 'appends' for adding new data. The flexibility of a NoSQL database will allow us to manage structurally different data from various sources (for example, for the various 'stores' that record sales), offering greater flexibility in data management.

- **System requirements.**

Shared-Nothing Architecture: The shared-nothing architecture is suitable for distributed systems, where the workload can be distributed across multiple nodes without the need to share significant resources. This architecture also facilitates task parallelization, allowing analytical operations to be executed concurrently on separate nodes.

Block-Based Partitioning: Block-based partitioning is the best choice for batch operations that require efficient reading of substantial volumes of data. It is an excellent way to distribute data across various disks, ensuring a balanced data load and preventing the occurrence of hot spots.

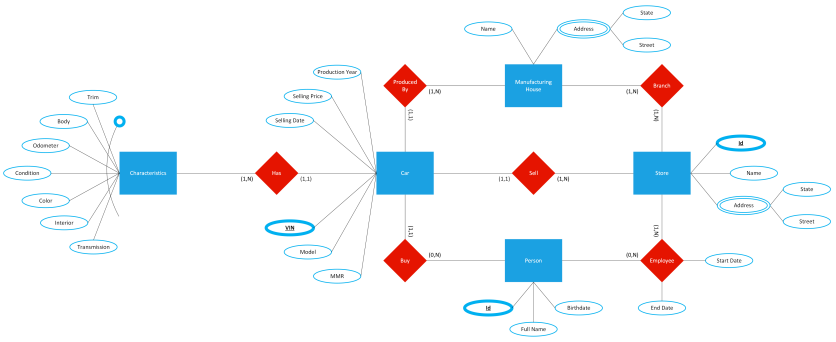
Automatic Rebalancing: Automatic rebalancing is essential, especially when dealing with large amounts of data. It ensures that the workload is evenly distributed among the system nodes, avoiding potential congestion points.

Replication with Asynchronous Write Propagation: Data replication is critical for distributed systems. In our analytical context, where we extract information from substantial volumes of data, asynchronous propagation of write operations is appropriate for our case. This method helps minimize the impact of write actions on reads, as writes occur less frequently than reads in our system. The choice of Single-Leader replication, combined with asynchronous write propagation, proves to be an advantageous configuration. The occurrence of updates within our system can be considered negligible. Therefore, this configuration helps us maintain efficiency while preserving data integrity.

Reflection on the CAP Theorem: In a distributed system, the CAP Theorem states that it is not possible to guarantee consistency, availability, and partition tolerance simultaneously. In our case, we align ourselves with the AP (Availability and Partition Tolerance) aspect of the CAP Theorem, giving preference to the system's availability. Partitioning is a crucial feature in distributed systems, implying the system's ability to continue functioning and reach every node in the network even in the event of failures. We have chosen to prioritize availability over consistency since update and write operations occur less frequently than data readings, and the availability property becomes crucial in situations of high data readings. In front of this two different approaches : ACID (Atomicity, Coherence, Isolation, Durability) and BASE (Basically Available, Soft State, Eventually Consistent) models. . The ACID model guarantees immediate consistency. However, in an analytical environment, where we prioritize the efficiency of read operations over immediate consistency, the BASE model is more suitable.

3 Conceptual Schema Design

- **Design of a conceptual schema.**



Here is a small summary of each entity and its attributes:

- **Entity**
  - **Car**
    - **Production Year:** Indicates the manufacturing year of the automobile.
    - **Selling Price:** The actual selling price of the automobile.
    - **Selling Date:** The date when the automobile was sold.
    - **VIN (Vehicle Identification Number):** A unique identification number assigned to the vehicle.
    - **Model:** The specific model of the automobile.
    - **MMR (Manheim Market Report):** The estimated value of the automobile according to the Manheim Market Report.
  - **Characteristics** (Weak Entity, provides more details about the car)
    - **Trim:** Denotes the level of trim or configuration of the automobile.
    - **Body:** Describes the body type of the automobile (e.g., sedan, coupe, SUV).
    - **Odometer:** The mileage or distance traveled by the vehicle.
    - **Condition:** Indicates the general condition of the automobile.
    - **Color** The exterior color of the automobile.
    - **Interior:** Describes the interior color of the automobile.

        * **Transmission:** Specifies the type of transmission of the automobile (automatic, manual).
    - **Person**
        * **Id:** Its Id.
        * **Full Name:** Indicates his/her full name.
        * **Year:** The date of his/her birthdate.
        * Note: Person should own at least one Car and/or be currently employed.
    - **Manufacturing House**
        * **Name:** Name of the brand or manufacturer.
        * **Address:** Specifies its location (with State and Street).
    - **Store**
        * **Id:** Id of the store
        * **Name:** Name of the store.
        * **Address:** Specifies its location (with State and Street).
- **Relations**
    - **Has:** Each Car has its Characteristics/Details. Relation many-to-one.
    - **Produced By** Each Manufacturing House produces Cars. Relation one-to-many.
    - **Buy** Each Person can buy some Cars. Relation zero-to-many, but each Car in the database has an owner.
    - **Sell** Each Store sells Cars. Relation one-to-many.
    - **Branch** Each Manufacturing House has at least one branch, at most shared with other companies. Relation many-to-many.
    - **Employee** Each Store has some Employees. Relation many-to-many.
        * **Start Date:** Indicates when the Employee began working for the Store
        * **End Date:** Specifies when the Employee stopped working for the Store.

4 Workload Identification

- **Identification of a workload with structurally different operations.**
- **Description of each workload operation in natural language.**

4.1 Queries

1 Sales Analysis for Top Selling Brands and Models:

- **Explanation:** This query provides insights into sales by showing the most popular brands and models. It counts the total number of sales for each brand-model combination and calculates the average selling price. This can be useful to understand which cars are in high demand and evaluate the financial performance of specific brands.
- **Attributes involved:** {Brand_Name, Model, SellingPrice}
- **Operators:** {group by, average, count}

2 Price Trends Analysis Over Time:

- **Explanation:** This query examines how car prices vary over time. It calculates the average selling prices for each year, allowing you to identify market trends in the automotive industry. This analysis is valuable for making informed financial decisions and predicting possible price changes.
- **Attributes involved:** {ProductYear, SellingPrice}
- **Operators:** {group by, average}

3 Price Distribution Analysis Based on Vehicle Condition:

- **Explanation:** This query analyzes how a vehicle's condition influences selling prices. It calculates the median prices for each vehicle condition, helping you understand how the vehicle's quality impacts its market value. It is useful for evaluating the effect of the condition on sales profitability.
- **Attributes involved:** {Condition, SellingPrice}
- **Operators:** {group by, median}

4 Correlation Analysis between Mileage and Selling Price:

- **Explanation:** This query examines the relationship between a vehicle's mileage and its selling price. It calculates the average selling prices based on different mileage ranges, helping you understand how the distance traveled influences the market value of a vehicle. This analysis can be useful for making financial decisions based on the correlation between mileage and price.
- **Attributes involved:** {Odometer, SellingPrice}
- **Operators:** {group by, average}

5 Price Distribution by Transmission Type:

- **Explanation:** This query examines price differences between automatic and manual vehicles, providing insights into the distribution of prices based on transmission type. This can provide some interesting data on customer interests.
- **Attributes involved:** {SellingPrice, Transmission}
- **Operators:** {group by, average}

6 Number of Sales by Brand for each Year:

- **Explanation:** This query offers an overview of sales, breaking down the data by year and brand. It provides valuable insights into the sales performance of each brand over different years.
- **Attributes involved:** {ProductionYear, Brand_Name}
- **Operators:** {group by, count}

7 Correlation between MMR and Selling Price:

- **Explanation:** This query evaluates the relationship between the estimated market value (MMR) and the selling price for a given model, providing valuable insights into pricing dynamics and the perception of value in the market.
- **Attributes involved:** {MMR, SellingPrice, Model}
- **Operators:** {group by, subtraction}

8 Sales by Body Type and Condition:

- **Explanation:** This query analyzes sales based on the body type and condition of the vehicle. By examining sales trends across different body types and conditions, businesses can gain insights into consumer preferences and market demand, informing strategic decisions related to inventory management, pricing strategies, and marketing efforts.
- **Attributes involved:** {Body, Condition, SellingPrice}
- **Operators:** {group by, average}

9 Total Sales Count:

- **Explanation:** This query counts the total number of sales in the dataset.
- **Attributes involved:** {VIN}
- **Operators:** {count}

10 Average Selling Price:

- **Explanation:** This query calculates the average selling price of all cars.
- **Attributes involved:** {SellingPrice}

- **Operators:** {average}

11 Top 5 Bestselling Brands:

- **Explanation:** This query identifies the five brands with the highest number of sales. This allows us to determine which are the most successful brands in the period examined, giving valuable insights into market dynamics, enabling strategic decision-making and competitive analysis.
- **Attributes involved:** {Brand_Name, VIN}
- **Operators:** {count, group by, order by (descending), limit 5}

12 Count of Cars in New Condition:

- **Explanation:** This query counts how many cars in the dataset are in perfect/new condition.
- **Attributes involved:** {Condition}
- **Operators:** {count}

13 Profit by Store for a Given Year:

- **Explanation:** For a given year, this query lists the profits for each store calculated as the difference between the selling price and MMR (estimated market value). This is useful for monitoring the earnings of the various stores over different years, aiding in strategic decision-making and resource allocation.
- **Attributes involved:** {ProductYear, Store_id, SellingPrice, MMR}
- **Operators:** {group by, average, subtraction}

14 Monthly Sales for a Given Store:

- **Explanation:** This query determines, given a store and a particular year, the lists of monthly sales. As with the previous query, this is useful for tracking the store earnings.
- **Attributes involved:** {Store_id, SellingDate}
- **Operators:** {group by, count, order by (descending)}

15 Average Buyer Age:

- **Explanation:** This query calculates the average age of buyers (so must have at least one car). This allows to make more targeted choices based on customer interests.
- **Attributes involved:** {Birthdate, VIN}
- **Operators:** {avg}

16 List of Cars Sold by Color (for a given model):

- **Explanation:** This query lists cars sold in order of popularity based on color for a given model, allowing sellers to direct production and marketing strategies accordingly.
- **Attributes involved:** {Model, Color}
- **Operators:** {group by, order by (descending), count}

17 Average Age by Condition:

- **Explanation:** This query calculates the average age of buyers who have purchased a car with a specific condition. It provides insights into the distribution of ages based on different vehicle conditions.
- **Attributes involved:** {Person_id,Birthday, Condition}
- **Operators:** {count, group by}

18 Affiliation of a Buyer given a Brand:

- **Explanation:** For a person who bought a car of a specific brand, this query determines if the buyer works for a branch or store that produces the same brand. This query provides crucial information about the customer's potential influence or preference for the brand.
- **Attributes involved:** {Person_id, Brand_Name (BranchEmployee), Brand_Name (PBBuy)}
- **Operators:** {innerjoin}

5 Aggregate-Oriented Design

- **Utilization of the aggregate-oriented to design aggregates.**

In the current stage of database development, we are engaged in the logical design, a pivotal step that transforms the conceptual schema and its associated workload into a logical schema. The approach to logical design embraces an aggregate-oriented model typical in NoSQL environments. We commence with the design of a meta-logical schema based on a meta-notation reminiscent of a JSON-like schema.
Subsequently, we translate this meta-logical schema into the aggregate-oriented model provided by the NoSQL system in use. A critical aspect of this translation is the identification of appropriate aggregates to optimize the queries in the workload.

5.1 Analysis of Queries

We conducted a thorough analysis of each query in our workload, clearly identifying the following components:

- **Entity (E):** For each query Q, we determined the entity to use as an aggregate.

- **Entity Set for Selection (LS):** We identified the set of entities used for selecting data to be returned as the query result. We focused on the WHERE clause of an equivalent SQL query, identifying the involved attributes and the paths connecting them to the main entity (E).
- **Entity Set for Projection (LP):** We determined the set of entities used for projecting data in the query response, corresponding to the SELECT clause of an equivalent SQL query, identifying the involved attributes and the paths connecting them to the main entity (E).

Now we can define each query as $Q(E, LS, LP)$, where E represents the aggregate, LS is the set of entities for selection, and LP is the set of entities for projection. This methodology is an important step for an efficient design of aggregates based on the workload's needs.

Q1
- $E = ManufacturingHouse$
- $LS = []$
- $LP = [ManufacturingHouse(Name)\_!, Car(Model, SellingPrice)\_PB]$

Q2
- $E = Car$
- $LS = []$
- $LP = [Car(ProductionYear, SellingPrice)\_!]$

Q3
- $E = Characteristics$
- $LS = []$
- $LP = [Characteristics(Condition)\_!, Car(SellingPrice)\_Has]$

Q4
- $E = Characteristics$
- $LS = []$
- $LP = [Characteristics(Odometer)\_!, Car(SellingPrice)\_Has]$

Q5
- $E = Characteristics$
- $LS = []$
- $LP = [Characteristics(Transmission)\_!, Car(SellingPrice)\_Has]$

Q6
- $E = Car$
- $LS = [Car(ProductionYear)\_!, ManufacturingHouse(Name)\_PB]$
- $LP = [Car(VIN)\_!]$

Q7
- $E = Car$
- $LS = [Car(Model)\_!]$
- $LP = [ManufacturingHouse(Name)\_PB, Car(SellingPrice, MMR)\_!]$

Q8
- $E = Characteristics$

- $LS = [Characterstics(Body)\_!]$
- $LP = [Characteristics(Condition)\_!, Car(SellingPrice)\_Has]$

Q9
- $E = Car$
- $LS = []$
- $LP = [Car(VIN)\_!]$

Q10
- $E = Car$
- $LS = []$
- $LP = [Car(SellingPrice)\_!]$

Q11
- $E = ManufacturingHouse$
- $LS = []$
- $LP = [ManufacturingHouse(Name)\_!, Car(VIN)\_PB]$

Q12
- $E = Characteristics$
- $LS = [Characteristics(Condition)\_!]$
- $LP = [Car(VIN)\_Has]$

Q13
- $E = Car$
- $LS = [Car(ProductionYear)\_!]$
- $LP = [Store(Id)\_Sell, Car(SellingPrice, MMR)\_!]$

Q14
- $Eh = Store$
- $LS = [Store(Id)\_!, Car(SD.year)\_Sell]$
- $LP = [Car(SD.month)\_Sell]$

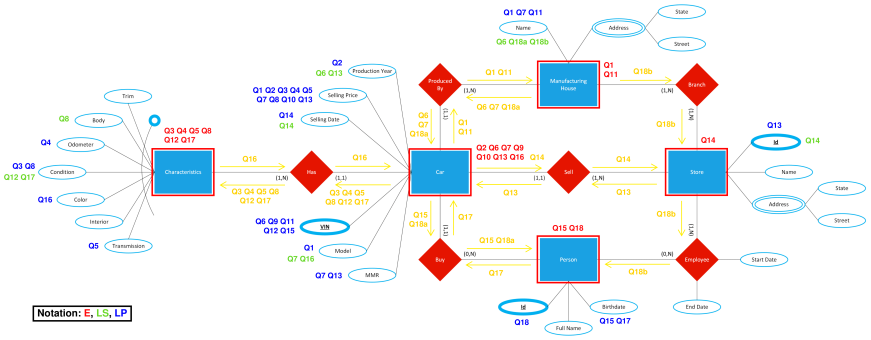
Q15
- $E = Person$
- $LS = []$
- $LP = [Person(Birthdate)\_!, Car(VIN)\_Buy]$

Q16
- $E = Car$
- $LS = [Car(Model)\_!]$
- $LP = [Characteristics(Color)\_Has]$

Q17
- $E = Characteristics$
- $LS = [Characteristics(Condition)\_!]$
- $LP = [Person(Birthdate)\_BuyHas]$

Q18
- $E = Person$
- $LS = [ManufacturingHouse(Name)\_PBBuy, ManufacturingHouse(Name)\_BranchEmployee]$
- $LP = [Person(id)\_!]$

5.2 Annotated E-R Diagram



5.3 Aggregates

1. **Aggregate Manufacturing House (MH)**

```
MH: {
    Name,
    Car: [{
        VIN,
        SellingPrice,
        Model
    }]
}
```

2. **Aggregate Characteristics**

```
Characteristics: {
                    Trim,
                    Body,
                    Odometer,
                    Condition,
                    Color,
                    Interior,
                    Transmission,
                      Car: [{
                              VIN,
                              SellingPrice,
                              Person_Birthdate
                            }]
                    }
```

3. **Aggregate Car**

```
Car: {
      VIN,
      ProductYear,
      SellingPrice,
      Model,
      Color,
      MMR,
      MH_Name,
      Store_id
    }
```

4. **Aggregate Person**

```
Person: {
        id,
        Birthdate,
           Car: [{
                 VIN,
                 MH_name
                }],
            Store: [{
                   Branch: [{
                           MH_Name
                           }]
                   }]
        }
```

5. **Aggregate Store**

```
Store: {
        id,
        Sell: [{
                Car_SellingDate
               }]
        }
```

6 Choosing Between MongoDB and Cassandra

- **Design of a schema for the selected system.**
- **Specification of each operation of the workload in the language supported by the selected system.**

After careful analysis, we settled on MongoDB as our database solution, even though we recognize that it could cover all the properties we think our application could cover. MongoDB aligns well with our *analytics scenarios* and excels in scenarios with *read-intensive operations*.

In contrast, Cassandra is better suited for *point queries* and focuses on *transactional capabilities*, which is not ideal for us. Cassandra would require the use of indexes, allowed filters, or passing intermediate values between the application and logical layers for many queries, making it less suitable for its intended use. While MongoDB doesn't have all the features we want, we consciously chose it over Cassandra due to its better alignment with our workload needs.

6.1 Schema for the MongoDB System.

1. **Aggregate Manufacturing House (MH)**

```
MH: {
    _id,
    Name,
    Car: [{
           VIN,
           SellingPrice,
           Model
          }]
    }

db.mh.createIndex({"Name": 1}, {unique: true})
db.adminCommand({shardCollection: "db.mh",
    key: {"Name": 1},
    field: "hashed" })
```

2. **Aggregate Characteristics**

```
Characteristics: {
                  _id,
                  Trim,
                  Body,
                  Odometer,
                  Condition,
                  Color,
                  Interior,
                  Transmission,
                  Car: [{
                           VIN,
                           SellingPrice,
                           Person_Birthdate
                        }]
                  }
```

```
db.characteristics.createIndex({"Trim": 1, "Body": 1, "Odometer
": 1, "Condition": 1, "Color": 1, "Interior": 1, "Transmission
": 1}, {unique: true})
db.adminCommand({shardCollection: "db.mh",
    key: {"Trim": 1, "Body": 1, "Odometer": 1, "Condition": 1,
"Color": 1, "Interior": 1, "Transmission": 1},
    field: "hashed" })
```

3. **Aggregate Car**

```
Car: {
        _id,
        VIN,
        ProductYear,
        SellingPrice,
        Model,
        Color,
        MMR,
        MH_Name,
        Store_id
    }

db.car.createIndex({"VIN": 1}, {unique: true})
db.adminCommand({shardCollection: "db.car",
    key: {"VIN": 1},
    field: "hashed"})
```

4. **Aggregate Person**

```
Person: {
        \_id,
        Birthdate,
        Car: [{
                VIN,
                MH_name
              }],
      Store: [{
                Branch: [{
                           MH_Name
                         }]
              }]
        }
```

5. **Aggregate Store**

```
Store: {
```

```
_id,
Sell: [{
         Car_SellingDate
       }]
}
```

6.2 Query for the MongoDB System.

The reader should note that placeholders are recognizable by the fact that they always start with "input...".

Q1
```
db.mh.aggregate([{$unwind: "$Car"},
        {$group: {
         _id: {
          "Brand": "$Name",
          "CarModel": "$Car.Model"
         },
         "AvgPrice": {$avg: "$Car.SellingPrice"},
         "Count": {$sum: 1}
         }
        }
        {$project: {
         _id: 0,
         "Brand": "$_id.Brand",
         "CarModel": "$_id.CarModel",
         "AvgPrice": 1,
         "Count": 1
         }
        }]);
```

Q2
```
db.car.aggregate([{$group: {
         _id: {"ProductionYear": "$ProductionYear"},
         "AvgPrice": {$avg: "$SellingPrice"},
         }
        },
        {$project: {
         _id: 0,
         "ProductionYear": "$_id.ProductionYear",
         "AvgPrice": 1
         }
        }]);
```

Q3
```
db.characteristics.aggregate([{$group: {
            _id: {"Condition": "$Condition"},
            "MedianPrice": {$median: {
```

```
            input: "$Car.SellingPrice",
            method: "approximate"}},
         }
        },
        {$project: {
         _id: 0,
         "Condition": "$_id.Condition",
         "MedianPrice": 1
         }
        }]);
```

Q4
```
db.characteristics.mapReduce(function() {
         var odometerInterval = Math.floor(this.Odometer / 2500) *
      2500;
         this.Car.forEach(function(car) {
          emit(odometerInterval, car.SellingPrice);
         });
        },
        function(key, values) {
         return Array.avg(values);
        });
```

Q5
```
db.characteristics.aggregate([{$unwind: "$Car"},
        {$group: {
         _id: {"Transmission": "$Transmission"},
         "AvgPrice": {$avg: "$Car.SellingPrice"},
         }
        },
        {$project: {
         _id: 0,
         "ProductionYear": "$_id.Transmission",
         "AvgPrice": 1
         }
        }]);
```

Q6
```
db.car.aggregate([{$group: {
        _id: {
         "ProductionYear": "$ProductionYear",
         "Brand": "$MH_Name",
        }
        "Sells": {$sum: 1},
        }
       },
       {$project: {
        _id: 0,
        "ProductionYear": "$_id.ProductionYear",
```

```
 "Brand": "$_id.Brand",
 "Sells": 1
 }
}]);
```

Q7
```
db.car.aggregate([{$match: {"$Model": inputModel}},
{$group: {
 _id: {"Brand": "$MH_Name"},
 avgSP: {$avg: "$selling price"},
 avgMMR: {$avg: "$MMR"}
 }
},
{$project: {
 _id: 0,
 "Brand": "$_id.Brand",
 "ProfitMargin": {$subtract: ["$avgMMR", "$avgSP"]}
}
}]);
```

Q8
```
db.characteristics.aggregate([{$match:{"$Body": inputBody}},
   {$unwind: "$Car"},
   {$group: {
    _id: {"Condition": "$Condition"},
    "avgPrice": {$avg: "$Car.SellingPrice"},
    }
   },
   {$project: {
    _id: 0,
    "Condition": "$_id.Condition",
    "avgPrice": 1
    }
   }]);
```

Q9
```
db.car.count();
```

Q10
```
db.car.find({}, {_id: 0, "Avg": {$avg: "$SellingPrice"}});
```

Q11
```
db.mh.find({}, {_id: 0, Name: 1, "Sells": {$size: "$Car"}}).sort({"
    Sells": -1}).limit(5);
```

Q12
```
db.characteristics.find({Condition: 5.0}).count();
```

Q13
```
db.car.aggregate([{$match: {"$ProductionYear": inputYear}},
        {$group: {
         _id: {"Store_id": "$Store_id"},
         avgSP: {$avg: "$SellingPrice"},
         avgMMR: {$avg: "$MMR"}
         }
        },
        {$project: {
         _id: 0,
         "Store_id": "$_id.Store_id",
         "ProfitMargin": {$subtract: ["$avgMMR", "$avgSP"]}
        }
        }]);
```

Q14
```
db.store.aggregate([{$unwind: "$Sell"},
         {$match: {
          $expr: {
           $eq: [{"SellingYear": {$year: "$Sell.Car_SellingDate"}},
        inputYear]
           }
          }
         },
         {$group: {
          "Month": {$month: "$Sell.Car_SellingDate"},
          "Count": {$sum: 1}
          }
         }]);
```

Q15
```
db.person.find({$nor: [
         {Car: {$exists: false}},
         {Car: {$size: 0}}]
         },
        {_id: 0,
        "AvgBirthdate": {$avg: {$divide: [{
         $subtract: [
          new Date(),
          {$toDate: "$Birthdate"}
         ]}, 365*24*60*60*1000]}}});
```

Q16
```
db.car.aggregate([{$match: {"$Model": inputModel}},
        {$group: {
         "Color": "$Color",
         "Count": { $sum: 1 }
         }
        },
        {$sort: {"Count": -1}
```

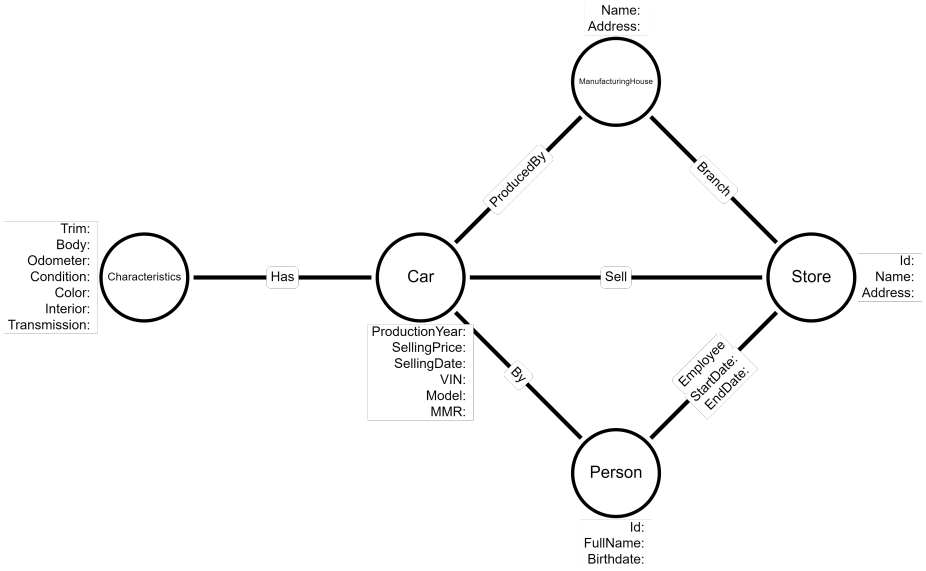
```
        }]);
```

Q17
```
db.characteristics.find({Condition: inputCondition},
          {$unwind: "$Car"},
          {_id: 0,
           "AvgBirthdate": {$avg:
            {$divide: [{
             $subtract: [
              new Date(),
              {$toDate: "$Car.Person_Birthdate"}
              ]}, 365*24*60*60*1000
             ]}}});
```

Q18
```
db.person.aggregate([{$match: {"$Car": {$in: [inputBrand]}}},
         {$unwind: "$Store"},
         {$match: {"$Store.Branch": {$in: [inputBrand]}}},
         {$project: {_id: 1}
         }]);
```

7 Neo4j Design

- **Schema design for Neo4j.**



- **Specification of each operation of the workload in Neo4j.**

The reader should note that placeholders are recognizable by the fact that they always start with "$input...".

Q1
```
MATCH (n:ManufacturingHouse)-[:ProducedBy]-(c:Car)
RETURN n.Name as Brand, c.Model as CarModel, round(AVG(c.
    SellingPrice), 2) as AvgPrice, COUNT(*) as Count;
```

Q2
```
MATCH (c:Car)
RETURN c.ProductionYear AS ProductionYear, round(AVG(c.SellingPrice
    ), 2) AS AvgPrice
ORDER BY ProductionYear DESC;
```

Q3
```
MATCH (ch:Characteristics)-[:Has]-(c:Car)
RETURN ch.Condition AS Condition, round(percentileDisc(c.
    SellingPrice, 0.5), 2) AS MedianPrice
ORDER BY Condition;
```

Q4
```
MATCH (ch:Characteristics)-[:Has]-(c:Car)
WITH FLOOR(ch.Odometer/2500)*2500 AS odometerInterval, c.
    SellingPrice AS SellingPrice
RETURN odometerInterval, round(AVG(SellingPrice), 2) AS
    AvgSellingPrice
ORDER BY odometerInterval;
```

Q5
```
MATCH (ch:Characteristics)-[:Has]-(c:Car)
RETURN ch.Transmission AS Transmission, round(AVG(c.SellingPrice),
    2) AS AvgPrice;
```

Q6
```
MATCH (c:Car)-[:ProducedBy]-(mh:ManufacturingHouse)
RETURN c.ProductionYear AS ProductionYear, mh.Name AS Brand, COUNT(
    c) AS Sells
ORDER BY ProductionYear DESC;
```

Q7
```
MATCH (c:Car {Model: $inputModel})-[:ProducedBy]-(mh:
    ManufacturingHouse)
RETURN mh.Name AS Brand, round(AVG(c.MMR - c.SellingPrice), 2) AS
    ProfitMargin;
```

Q8
```
MATCH (ch:Characteristics {Body: $inputBody})-[:Has]-(c:Car)
RETURN ch.Condition AS Condition, round(AVG(c.SellingPrice), 2) AS
    AvgPrice
ORDER BY Condition;
```

Q9
```
MATCH (c:Car)
RETURN COUNT(c) AS TotalCarsCount;
```

Q10
```
MATCH (c:Car)
RETURN round(AVG(c.SellingPrice), 2) AS AvgSellingPrice;
```

Q11
```
MATCH (mh:ManufacturingHouse)-[:ProducedBy]-(c:Car)
RETURN mh.Name, COUNT(c) AS Sells
ORDER BY Sells DESC
LIMIT 5;
```

Q12
```
MATCH (:Characteristics {Condition: 5.0})-[:Has]-(c:Car)
RETURN COUNT(c) AS Count;
```

Q13
```
MATCH (s:Store)-[:Sell]-(c:Car {ProductionYear: $inputYear})
RETURN s.id AS Store_id, round(AVG(c.MMR - c.SellingPrice), 2) AS
    ProfitMargin
ORDER BY ProfitMargin DESC;
```

Q14
```
MATCH (:Store {id: $inputStoreId})-[:Sell]-(c:Car)
WHERE datetime(c.SellingDate).year = $inputYear
RETURN datetime(c.SellingDate).month AS Month, COUNT(*) AS Count
ORDER BY Month;
```

Q15
```
MATCH (p:Person)-[:Buy]-(:Car)
RETURN round(AVG(datetime().year - datetime(p.Birthdate).year)) AS
    avgAge;
```

Q16
```
MATCH (:Car {Model: $inputModel})-[:Has]-(ch:Characteristics)
RETURN ch.Color AS Color, COUNT(*) AS CountC
ORDER BY CountC DESC;
```

Q17
```
MATCH (:Characteristics {Condition: $inputCondition})-[:Has]-(:Car)
    -[:Buy]-(p:Person)
RETURN round(AVG(datetime().year - datetime(p.Birthdate).year)) AS
    avgBirthdate;
```

Q18
```
MATCH (p:Person)-[:Buy]-(:Car)-[:ProducedBy]-(m:ManufacturingHouse
    {Name: $inputBrand})
MATCH (p)-[:Employee]-(:Store)-[:Branch]-(m)
RETURN p.id AS PersonId;
```

8 Discussion on System Selection

- **Motivation for choosing one system among MongoDB, Cassandra, and Neo4j.**

We examined various database options, focusing on MongoDB, Cassandra and Neo4j, to determine the best choice for our back-end system dedicated to financial analysis and sales monitoring in the automotive industry.

We aim to achieve availability and partition tolerance, following the CAP theorem. We are looking for a database that follows the BASIC principles. These databases are known to handle large amounts of data quickly, with high throughput, and do not exhibit no synchronization. It is essential to note that ACID databases, in contrast, focus on data consistency. We would like to prioritize system availability, even at the cost of immediate data consistency.

Reasons for not choosing MongoDB and Cassandra:

1. **MongoDB:**
    - Extracting information often required map-reduce and aggregation-based operations, resulting in complex data management due to numerous correlations. MongoDB was not chosen because, in this context, our queries were more oriented towards relationships between data. While we could have used map-reduce, we believe there was a better strategy available.
    - The second reason, as mentioned in the preface, concerns the choice to move towards a solution that places greater emphasis on availability.
2. **Cassandra:**
    - Cassandra adapts well to transactional scenarios. It does not meet analytical needs. This is because, although it follows the availability and partition principles of the CAP theorem, its focus on query performance introduces excessive restrictions on query executions. As

detailed in the previous points, we would have had to impose system forcing or index uses.

Reasons for choosing Neo4j:

1. **Performance in Analytical Queries:**
   - It has demonstrated superior performance in analytical queries, as much of them rely on efficiently searching relationships between different instances. The graph structure fits perfectly with our needs to explore data interconnections.
2. **Framework:**
   - To boost query speed and overall performance, explore alternative frameworks. For instance, you can distribute graph algorithms using Mazerunner, which is built on top of Spark, GraphX, and extended Neo4j instances.
3. **Graph Search Algorithms:**
   - It has well-established graph search algorithms, allowing us to extract detailed information and effectively respond to advanced queries.
4. **Intuitive Graph Representation:**
   - The graphical nature of data in Neo4j makes the representation of relationships easily intuitive, simplifying the exploration and visualization of complex connections.

In conclusion, Neo4j was chosen for its performance in analytical queries, efficient management of complex relationships, presence of well-established graph search algorithms, and the intuitive graphical representation of data. However, it should be noted that it does not currently support sharding, a feature we would consider advantageous. Alternatively, we could explore the use of specific frameworks or evaluate the possibility of adopting ArangoDB, which offers sharding solutions. The benefits of graphs are widely discussed in the followed paper: Graph Databases: Neo4j Analysis

9 System Configuration Details

- **Details on the system configuration needed for storing/processing data according to the chosen application.**

For the implementation the ideal choice would have been Neo4j ArangoDB, as it would also allow you to manage sharding, furthermore being based on C++ it allows you to obtain maximum performance.
For these characteristics alone it would be the perfect candidate for our domain, but unfortunately it is paid and therefore we had to stay with Neo4j AuraDB

(Community Edition). Another negative note is that, since we are using the free version, there are many limitations: There is a maximum number of nodes and relationships (which we solved by using the offline version, so we can use all our data, composed by the equivalent of 1M rows) and the number of settings that can actually be adjusted is relatively low.
Among the few adjustable settings, the one that interests us most is the role-based access control, as it allows us to manage which users are able to access and execute our queries (the workload would obviously be a sample). Among the non-adjustable ones, fortunately, the one about the type of system is already set correctly as asynchronous Master-slave (single-leader).

In our context, creating indexes would not be of much use, given that the workload is batch-type, but we wanted to implement them anyway, since there could be a benefit even if it's very small.

The only attributes that really interest us given the workload are the following, with their respective commands for index creation:

- **Car's Model [Car]**

```
CREATE INDEX IF NOT EXISTS FOR (c:Car) ON (c.Model);
```

- **Car's Body [Characteristics]**

```
CREATE INDEX IF NOT EXISTS FOR (ch:Characteristics) ON (ch.Body);
```

- **Car's Condition [Characteristics]**

```
CREATE INDEX IF NOT EXISTS FOR (ch:Characteristics) ON (ch.
    Condition);
```

- **Car's Prodution Year [Car]**

```
CREATE INDEX IF NOT EXISTS FOR (c:Car) ON (c.ProductionYear);
```

- **Store's id [Store]**

```
CREATE INDEX IF NOT EXISTS FOR (s:Store) ON (s.id);
```

- **Manufacturing House's Name [ManufacturingHouse]**

```
CREATE INDEX IF NOT EXISTS FOR (mh:ManufacturingHouse) ON (mh.Name)
    ;
```

While for the id of Car (VIN) and Person are not used as a selection attribute and therefore do not necessarily require the index, what should be for the year of SellingDate (Car) cannot be created, as it would require breaking down/splitting it, a change that is feasible but is not necessary and would not be sufficient to justify it in a real scenario.

10 Logical Schema Creation in System S

- **Creation of the logical schema in the selected system.**

The reader should note that the placeholders are all "highlighted" by double quotation marks (").

10.1 Create Nodes

- **Car**

```
CREATE CONSTRAINT IF NOT EXISTS FOR (c:Car) REQUIRE (c.VIN) IS
    UNIQUE;
CREATE (:Car {ProductionYear: "year", SellingPrice: "price",
    SellingDate: "date", VIN: "vin", Model: "model", MMR: "mmr"})
```

- **Characteristics**

```
CREATE CONSTRAINT IF NOT EXISTS FOR (ch:Characteristics) REQUIRE (
    ch.Trim, ch.Body, ch.Odometer, ch.Condition, ch.Color, ch.
    Interior, ch.Transmission) IS UNIQUE;
CREATE (:Characteristics {Trim: "trim", Body: "body", Odometer: "od
    ", Condition: "cond", Color: "color", Interior: "color",
    Transmission: "tra"})
```

- **Person**

```
CREATE CONSTRAINT IF NOT EXISTS FOR (p:Person) REQUIRE (p.id) IS
    UNIQUE;
CREATE (:Person {id: "id", FullName: "fullname", Birthdate: "date
    "})
```

- **ManufacturingHouse**

```
CREATE CONSTRAINT IF NOT EXISTS FOR (mh:ManufacturingHouse) REQUIRE
     (mh.Name) IS UNIQUE;
CREATE (:ManufacturingHouse {Name: "name", Address: "address"})
```

- **Store**

```
CREATE CONSTRAINT IF NOT EXISTS FOR (s:Store) REQUIRE (s.id) IS
    UNIQUE;
CREATE (:Store {id: "id", name: "name", Address: "address"})
```

10.2 Create Relationships

- **Has**

```
MATCH (c:Car {VIN: "vin"}), (ch:Characteristics {Trim: "trim"})
MERGE (c)-[:Has]->(ch)
```

- **ProducedBy**

```
MATCH (mh:ManufacturingHouse {Name: "name"}), (c:Car {VIN: "vin"})
MERGE (mh)<-[:ProducedBy]-(c)
```

- **Buy**

```
MATCH (p:Person {id: "id"}), (c:Car {VIN: "vin"})
MERGE (p)-[:Buy]->(c)
```

- **Sell**

```
MATCH (s:Store {id: "id"}), (c:Car {VIN: "vin"})
MERGE (s)-[:Sell]->(c)
```

- **Branch**

```
MATCH (mh:ManufacturingHouse {Name: "name"}), (s:Store {id: "id"})
MERGE (mh)-[:Branch]->(s)
```

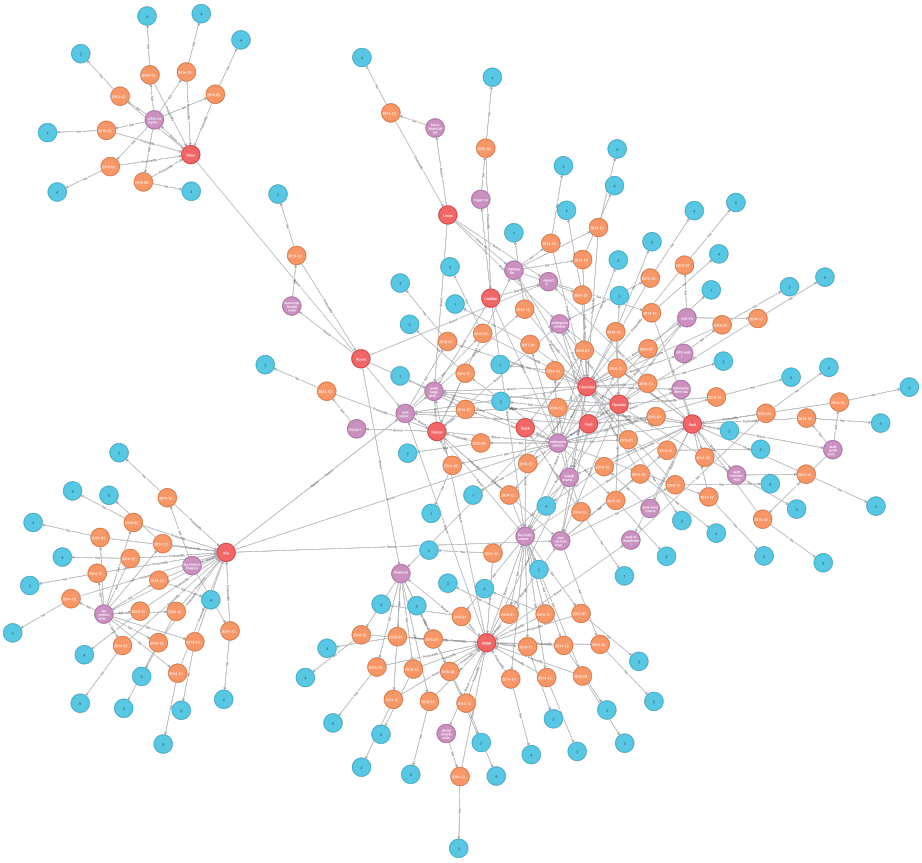
- **Employee**

```
MATCH (s:Store {id: "id"}), (p:Person {id: "id"})
MERGE (s)<-[:Employee {StartDate: "date", EndDate: "date or null
    "}]-(p)
```

11 Dataset Instance Creation

- **Creation of an instance of the schema in the selected system using a relevant dataset.**

Below is a screen of a part of the implemented graph (limited to 300 nodes by the system).



12 Implementing the Workload in System S

- **Implementation of the workload in the selected system.**

In general, even though we were dealing with a total number of nodes equal to approximately 1M and many more relationships, all queries completed the operation without taking more than a couple of seconds (literally).

Q1

```
neo4j$ MATCH (n:ManufacturingHouse)-[:ProducedBy]-(c:Car) RETURN n.Name as Brand, c.Model as CarModel, round(AVG(c.SellingPric…
```

	Brand	CarModel	AvgPrice	Count
1	"Acura"	"MDX"	17096.01	1334
2	"Acura"	"TL"	11952.25	1765
3	"Acura"	"ZDX"	26059.68	31
4	"Acura"	"RSX"	4753.72	121
5	"Acura"	"TSX"	14011.64	915
6	"Acura"	"CL"	1928.27	84
7	…	…	…	…

Started streaming 777 records after 2 ms and completed after 1174 ms.

Q2

```
neo4j$ MATCH (c:Car) RETURN c.ProductionYear AS ProductionYear, round(AVG(c.SellingPrice), 2) AS AvgPrice ORDER BY ProductionY…
```

	ProductionYear	AvgPrice
1	2015	25954.65
2	2014	19953.43
3	2013	17400.75
4	2012	16533.43
5	2011	15820.38
6	2010	12471.86
7	…	…

Started streaming 26 records after 9 ms and completed after 549 ms.

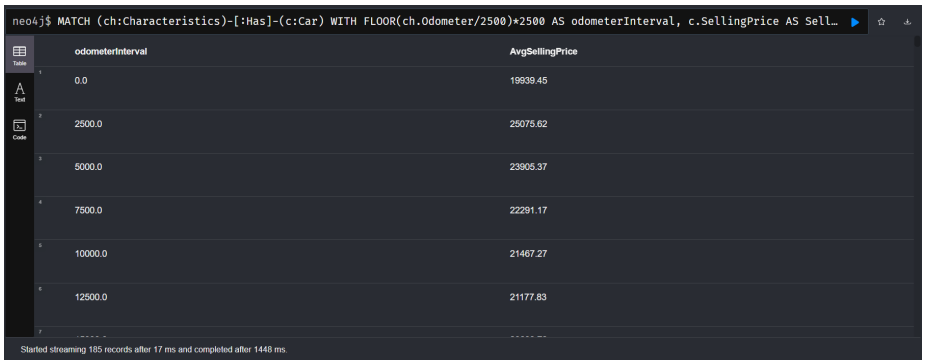
Q3

```
neo4j$ MATCH (ch:Characteristics)-[:Has]-(c:Car) RETURN ch.Condition AS Condition, round(percentileDisc(c.SellingPrice, 0.5), …
```

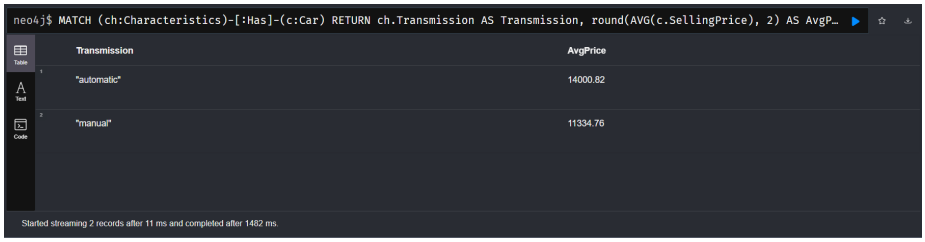
	Condition	MedianPrice
1	1	3500.0
2	2	7600.0
3	3	12500.0
4	4	17100.0
5	5	22000.0

Started streaming 5 records after 11 ms and completed after 1622 ms.

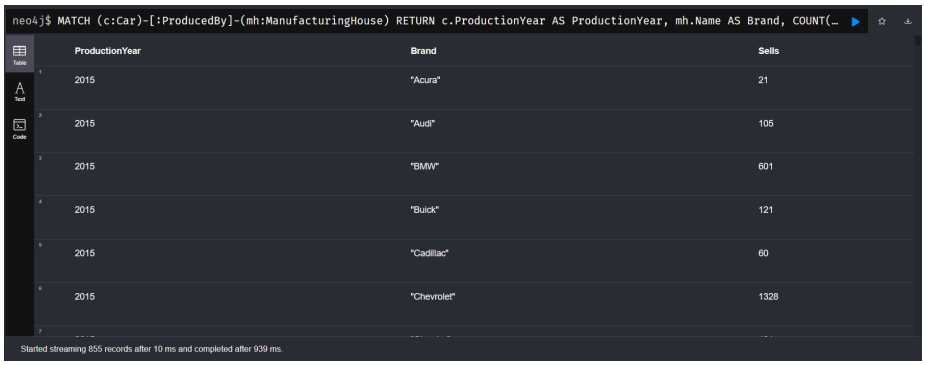
Q4


neo4j$ MATCH (ch:Characteristics)-[:Has]-(c:Car) WITH FLOOR(ch.Odometer/2500)*2500 AS odometerInterval, c.SellingPrice AS Sell…
Table
Text
Code
odometerInterval
AvgSellingPrice
0.0
19939.45
2500.0
25075.62
5000.0
23905.37
7500.0
22291.17
10000.0
21467.27
12500.0
21177.83
Started streaming 185 records after 17 ms and completed after 1448 ms.

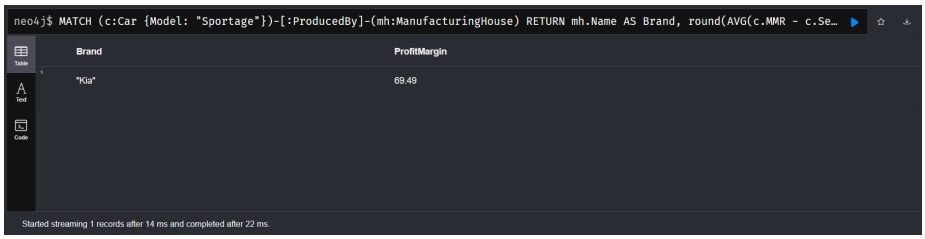
Q5


neo4j$ MATCH (ch:Characteristics)-[:Has]-(c:Car) RETURN ch.Transmission AS Transmission, round(AVG(c.SellingPrice), 2) AS AvgP…
Table
Text
Code
Transmission
AvgPrice
"automatic"
14000.82
"manual"
11334.76
Started streaming 2 records after 11 ms and completed after 1482 ms.

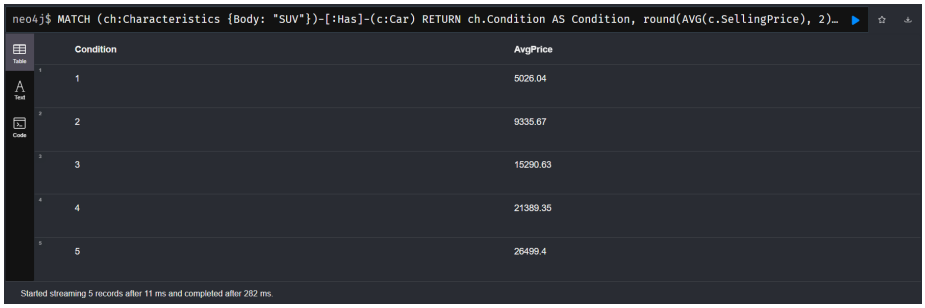
Q6


neo4j$ MATCH (c:Car)-[:ProducedBy]-(mh:ManufacturingHouse) RETURN c.ProductionYear AS ProductionYear, mh.Name AS Brand, COUNT(…
Table
Text
Code
ProductionYear
Brand
Sells
2015
"Acura"
21
2015
"Audi"
105
2015
"BMW"
601
2015
"Buick"
121
2015
"Cadillac"
60
2015
"Chevrolet"
1328
Started streaming 855 records after 10 ms and completed after 939 ms.

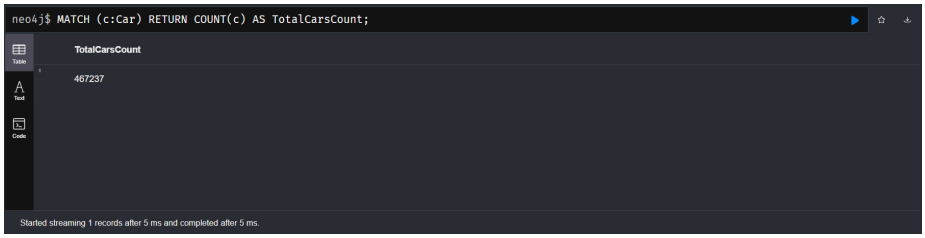
Q7


neo4j$ MATCH (c:Car {Model: "Sportage"})-[:ProducedBy]-(mh:ManufacturingHouse) RETURN mh.Name AS Brand, round(AVG(c.MMR - c.Se…
Table
Text
Code
Brand
ProfitMargin
1
"Kia"
69.49
Started streaming 1 records after 14 ms and completed after 22 ms.

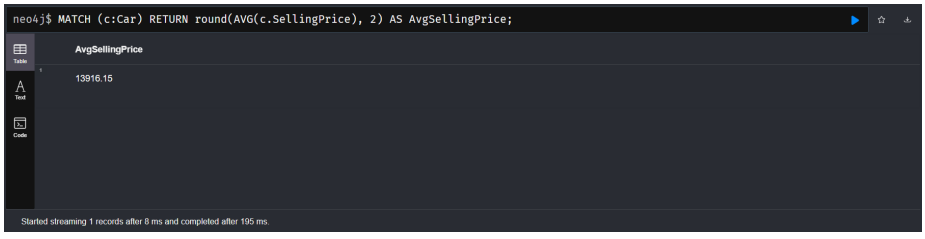
Q8


neo4j$ MATCH (ch:Characteristics {Body: "SUV"})-[:Has]-(c:Car) RETURN ch.Condition AS Condition, round(AVG(c.SellingPrice), 2)…
Table
Text
Code
Condition
AvgPrice
1
1
5026.04
2
2
9335.67
3
3
15290.63
4
4
21389.35
5
5
26499.4
Started streaming 5 records after 11 ms and completed after 282 ms.

Q9


neo4j$ MATCH (c:Car) RETURN COUNT(c) AS TotalCarsCount;
Table
Text
Code
TotalCarsCount
1
467237
Started streaming 1 records after 5 ms and completed after 5 ms.

Q10


neo4j$ MATCH (c:Car) RETURN round(AVG(c.SellingPrice), 2) AS AvgSellingPrice;
Table
Text
Code
AvgSellingPrice
1
13916.15
Started streaming 1 records after 8 ms and completed after 195 ms.

Q11

```
neo4j$ MATCH (mh:ManufacturingHouse)-[:ProducedBy]-(c:Car) RETURN mh.Name, COUNT(c) AS Sells ORDER BY Sells DESC LIMIT 5;
```

	mh.Name	Sells
1	"Ford"	80616
2	"Chevrolet"	51365
3	"Nissan"	47024
4	"Toyota"	34706
5	"Dodge"	26622

Started streaming 5 records after 10 ms and completed after 282 ms.

Q12

```
neo4j$ MATCH (:Characteristics {Condition: 5.0})-[:Has]-(c:Car) RETURN COUNT(c) AS Count;
```

	Count
1	8945

Started streaming 1 records after 8 ms and completed after 27 ms.

Q13

```
neo4j$ MATCH (s:Store)-[:Sell]-(c:Car {ProductionYear: 2015}) RETURN s.id AS Store_id, round(AVG(c.MMR - c.SellingPrice), 2) A…
```

	Store_id	ProfitMargin
1	5580537	25500.0
2	5590131	22500.0
3	5590358	16150.0
4	5580653	12700.0
6	5590147	11400.0
5	5584478	11200.0
7	-------	------

Started streaming 817 records after 11 ms and completed after 53 ms.

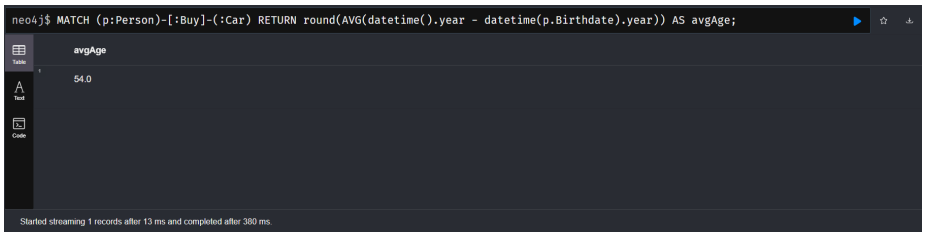
Q14

```
neo4j$ MATCH (:Store {id: 5580537})-[:Sell]-(c:Car) WHERE datetime(c.SellingDate).year = 2015 RETURN datetime(c.SellingDate).m…
```

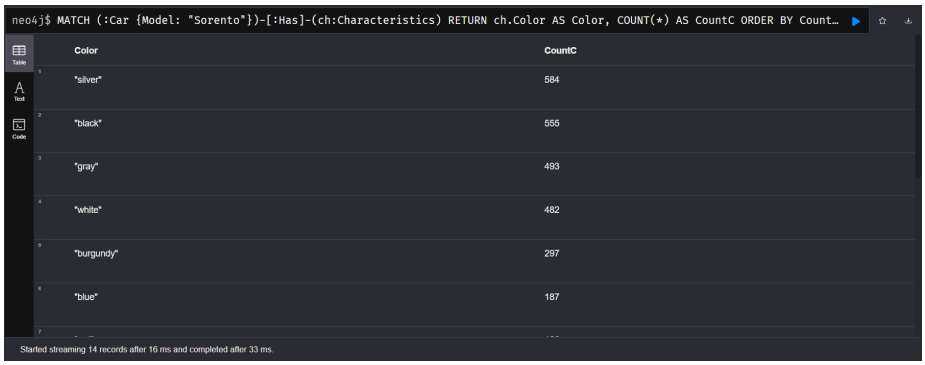
	Month	Count
1	3	1
2	4	2
3	5	2
4	6	7

Started streaming 4 records after 14 ms and completed after 16 ms.

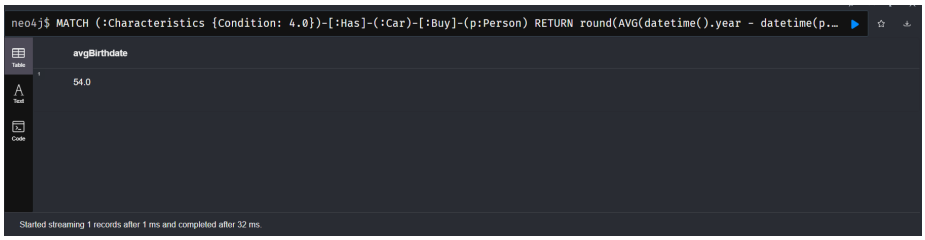
Q15



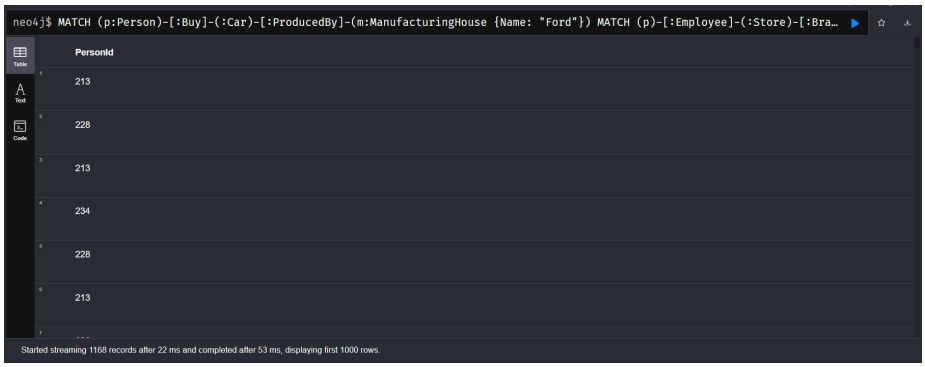
Q16



Q17



Q18


neo4j$ MATCH (p:Person)-[:Buy]-(:Car)-[:ProducedBy]-(m:ManufacturingHouse {Name: "Ford"}) MATCH (p)-[:Employee]-(:Store)-[:Bra…
Table
Text
Code
PersonId
1 213
2 228
3 213
4 234
5 228
6 213
Started streaming 1168 records after 22 ms and completed after 53 ms, displaying first 1000 rows.