ADM-Project

Edoardo Pozzi 4831619 - Andrea Alfieri 4829586 we both took the exam on 09/01/2024.

Proposed domain

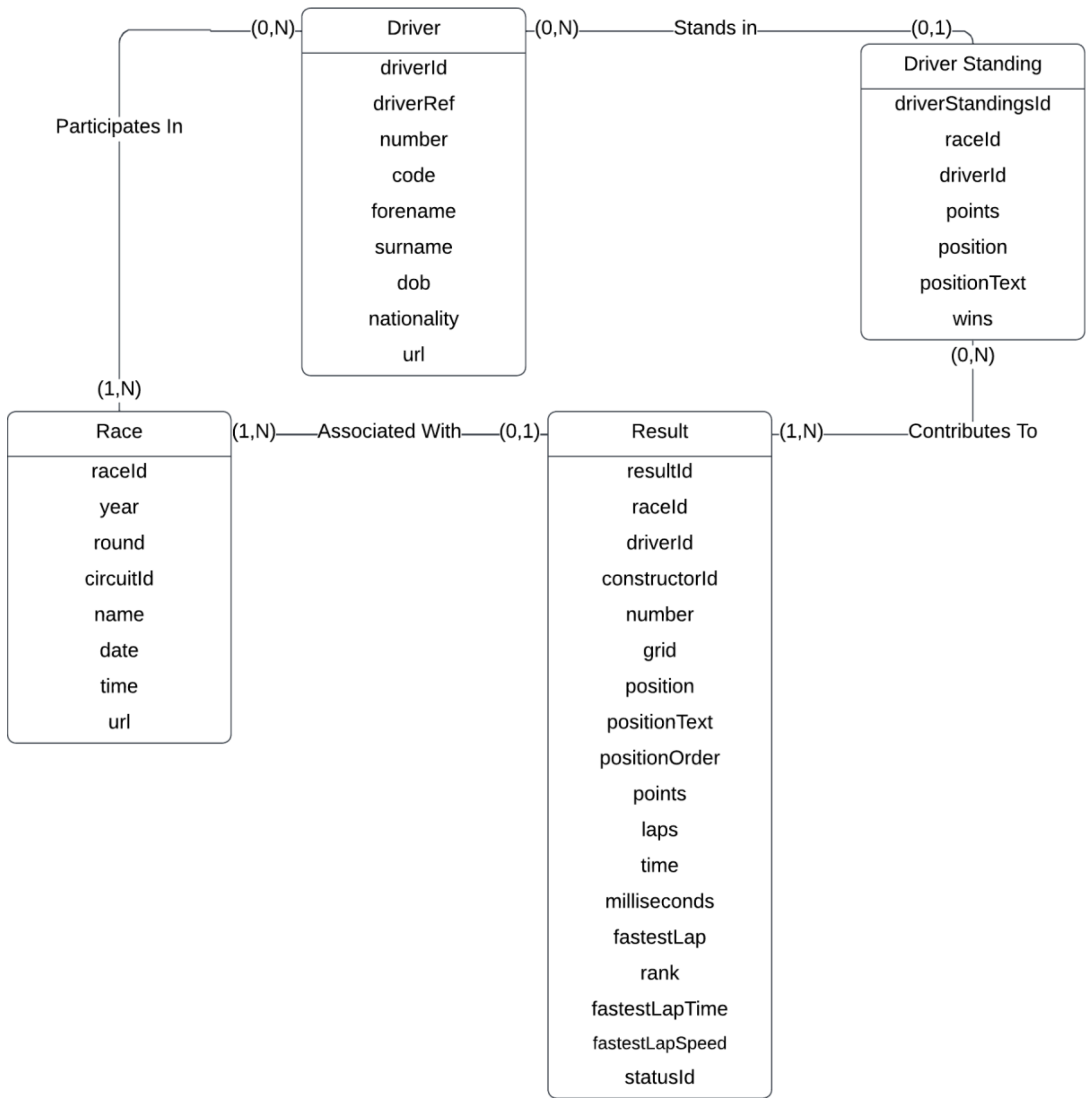
The proposed domain is a website regarding Formula One, the database consists of a list of drivers that participate in races held at grand Prix's. Based on the position in which each driver finishes a race a given number of points is assigned to him, a final standing is made based on the past results.

Users access the site to look at the results of a race, the standings of their favorite driver and the final standings of a driver.

Nature of the proposed application

The database is composed of the results of each grand Prix that are added after each race weekend.
Based on that information, we can say that the nature of the application is not write intensive since the write operations are done only once/twice a month depending on the number of races and do not add a large quantity of data. Most of the operations are read operations, for example to look at the results of a race or the standings of a driver.
Regarding the CAP theorem, we are looking for a system that is CP. The consistency is vital since the results should always be correct and no read errors are allowed; partitioning is also important since numerous users will connect to the website at the same time to look at the results and thanks to partitioning the latency can be reduced.

Conceptual schema



Workload

Query 1:
Given a driver surname, return the races he participated in and their respective results.
Query 2:
Given a race name and date, get surname and nationality of the driver who won that race.
Query 3:
Given a nationality, see its drivers' forename and surname along with their driver standing's points and position.
Query 4:
Given a driver surname and number, get their standings.
Query 5:
Get the forename of the first drivers in the leaderboard of the driver standings along with their results.

Aggregate-oriented design methodology.

Aggregates based on the workload.

Q1
E = Driver
LS = [Driver(surname) _!]
LP = [Race(raceId) _P, Result(resultId) _AP]

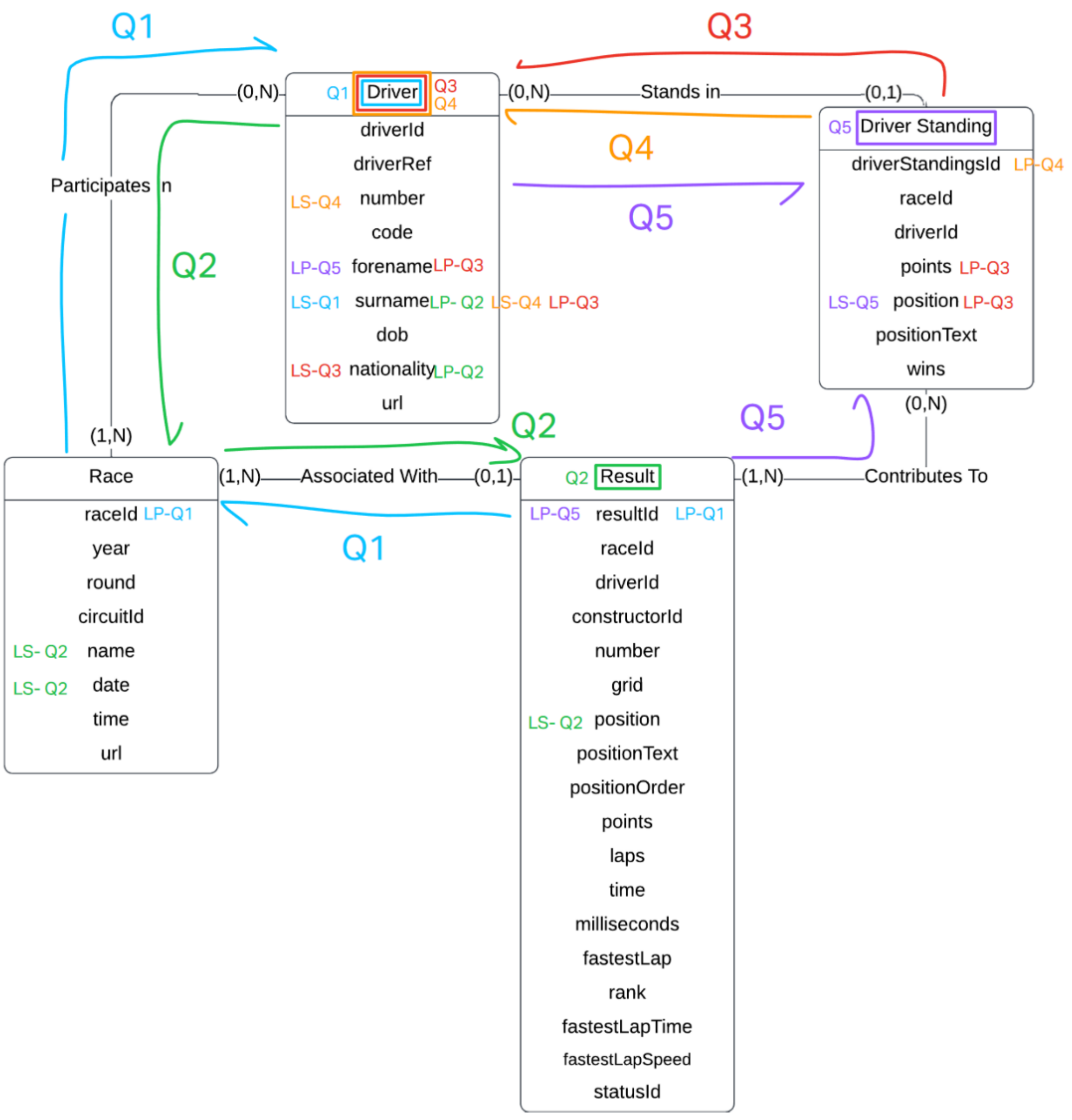
Q2
E = Result
LS = [Race(name, date) _A, Result(position) _!]
LP = [Driver(surname, nationality)_PA]

Q3
E = Driver
LS = [Driver(nationality) _!]
LP = [Driver(forename, surname) _!, DriverStanding(points, position) _S]

Q4
E = Driver
LS = [Driver(number, surname) _!]
LP = [DriverStanding(driverStadingsId) _S]

Q5
E = Driver Standing
LS = [Driver Standing(position) _!]
LP = [Driver(forename) _S , Result(resultId) _C]

Annotated schema



Final aggregate

Driver: {**driverId**, number, forename, surname, nationality, WhichStandsIn:[DriverStanding: {driverStandingsId, points, position}], ParticipatesIn: [{Race: {raceId, AssociatedWith: [Result: {resultId}]}}]}

Result: {**resultId**, position, name, date, surname, nationality}

DriverStanding: {**driverStandingId**, position, forename, resultId}

System choice

Motivation

We chose MongoDB since it is a CP Database and consistency is a vital aspect of our application and for the possibility to have nested attributes that we used to encapsulate in a single document different associations that were present in the relational model.

Partition key and indexes.

For aggregate Driver the partition key is set to surname and nationality:

```
db.adminCommand( { shardCollection: "db.Drivers", key: { surname: 1, nationality :1 }, field: "hashed" } )
```

For aggregate Result the partition key is set to name and date:

```
db.adminCommand( { shardCollection: "db.Constructors", key: { name: 1, date:1 }, field: "hashed" } )
```

For aggregate DriverStanding the partition key is set to position:

```
db.adminCommand( { shardCollection: "db.Results", key: { position: 1 }, field: "hashed" } )
```

Partition key choice is vital for efficient queries, based on that we choose the key that partitions the documents in the most effective way.

For driver there is a selection on the attribute “Surname”: having all the drivers with the same surname in the same node is vital for efficient queries so they don’t need to navigate through different nodes; the same principle is used to select the others.

For DriverStanding aggregate indexes have been created on fields position, driverStandingId:

```
db.DriverStanding.createIndex( {"position": 1}, {unique: false})
db.DriverStanding.createIndex( {"driverStandingId": 1}, {unique: true})
```

For Driver aggregate indexes have been created on fields surname, nationality, driverId:

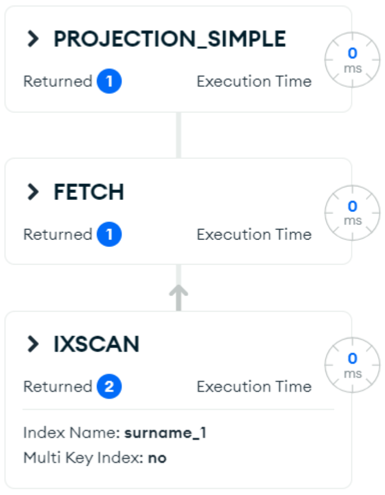
```
db.Driver.createIndex( {"surname": 1}, {unique: false})
db.Driver.createIndex( {"nationality": 1}, {unique: false})
db.Driver.createIndex( {"driverId": 1}, {unique: true})
```

For Result aggregate additional indexes have been created on fields resultId, date + name + position

```
db.Result.createIndex( {"resultId": 1}, {unique: true})
db.Result.createIndex( {"date": 1, "name": 1, "position": 1,}, {unique: false})
```

Indexes also help with the efficiency so we created some of them based on the workload.

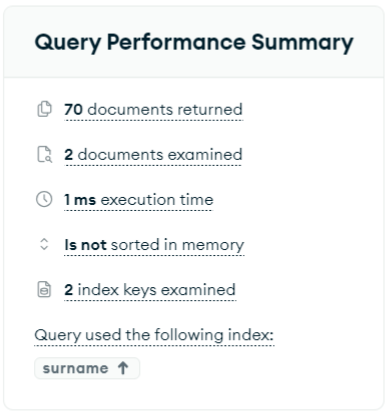
Below we can see the decrease in the number of document scans needed to complete the query.



PROJECTION_SIMPLE
Returned 1 Execution Time 0 ms

FETCH
Returned 1 Execution Time 0 ms

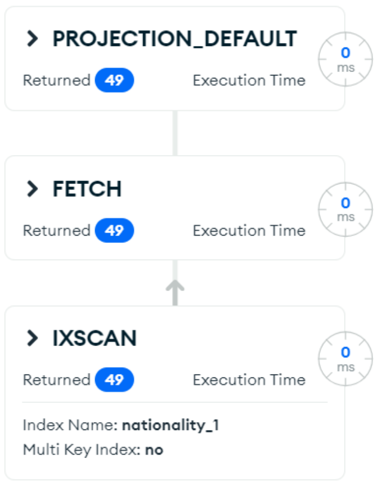
IXSCAN
Returned 2 Execution Time 0 ms
Index Name: **surname_1**
Multi Key Index: **no**



Query Performance Summary

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

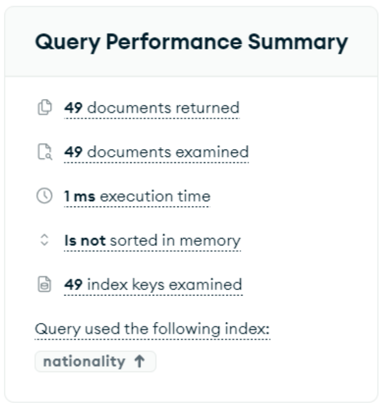
Query used the following index:
surname ↑



PROJECTION_DEFAULT
Returned 49 Execution Time 0 ms

FETCH
Returned 49 Execution Time 0 ms

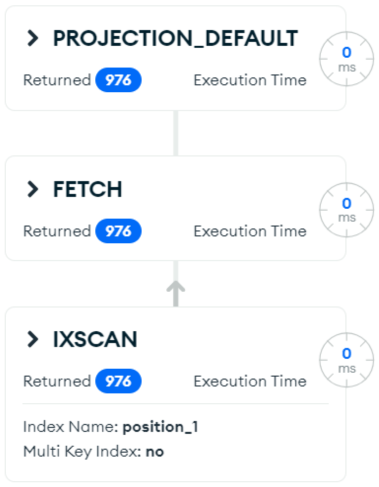
IXSCAN
Returned 49 Execution Time 0 ms
Index Name: **nationality_1**
Multi Key Index: **no**



Query Performance Summary

- **49** documents returned
- **49** documents examined
- **1 ms** execution time
- **Is not** sorted in memory
- **49** index keys examined

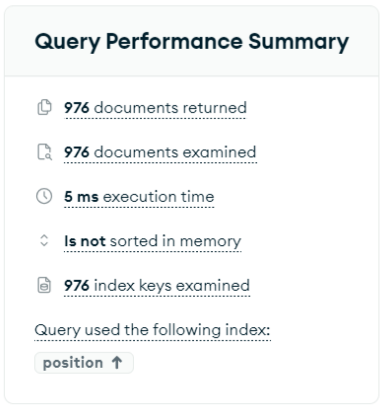
Query used the following index:
nationality ↑



PROJECTION_DEFAULT
Returned 976 Execution Time 0 ms

FETCH
Returned 976 Execution Time 0 ms

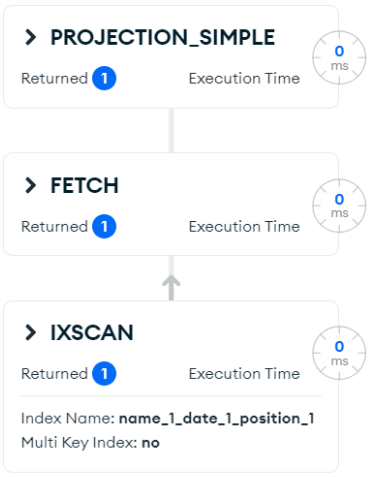
IXSCAN
Returned 976 Execution Time 0 ms
Index Name: **position_1**
Multi Key Index: **no**



Query Performance Summary

- **976** documents returned
- **976** documents examined
- **5 ms** execution time
- **Is not** sorted in memory
- **976** index keys examined

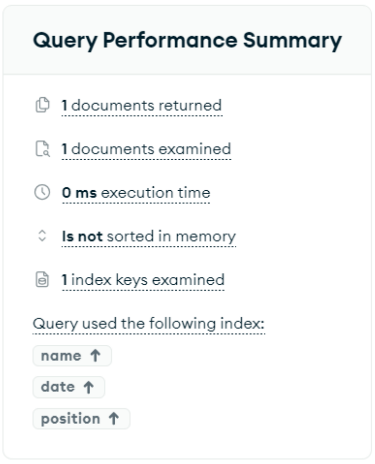
Query used the following index:
position ↑



PROJECTION_SIMPLE
Returned 1 Execution Time 0 ms

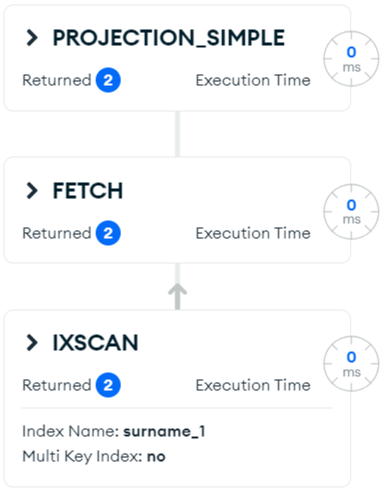
FETCH
Returned 1 Execution Time 0 ms

IXSCAN
Returned 1 Execution Time 0 ms
Index Name: **name_1_date_1_position_1**
Multi Key Index: **no**



Query Performance Summary

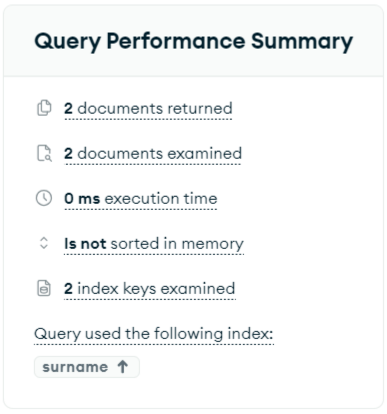
1 documents returned
1 documents examined
0 ms execution time
Is not sorted in memory
1 index keys examined
Query used the following index:
name ↑
date ↑
position ↑



PROJECTION_SIMPLE
Returned 2 Execution Time 0 ms

FETCH
Returned 2 Execution Time 0 ms

IXSCAN
Returned 2 Execution Time 0 ms
Index Name: **surname_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:
surname ↑

Queries in MongoDB

db.Driver.find({"surname": "YOUR_SELECTION"}, {ParticipatesIn: 1})

db.Result.find({"name": "YOUR_SELECTION", "date": ISODate("YOUR_ISO_DATE"), "position": 1}, {"surname": 1, "nationality": 1})

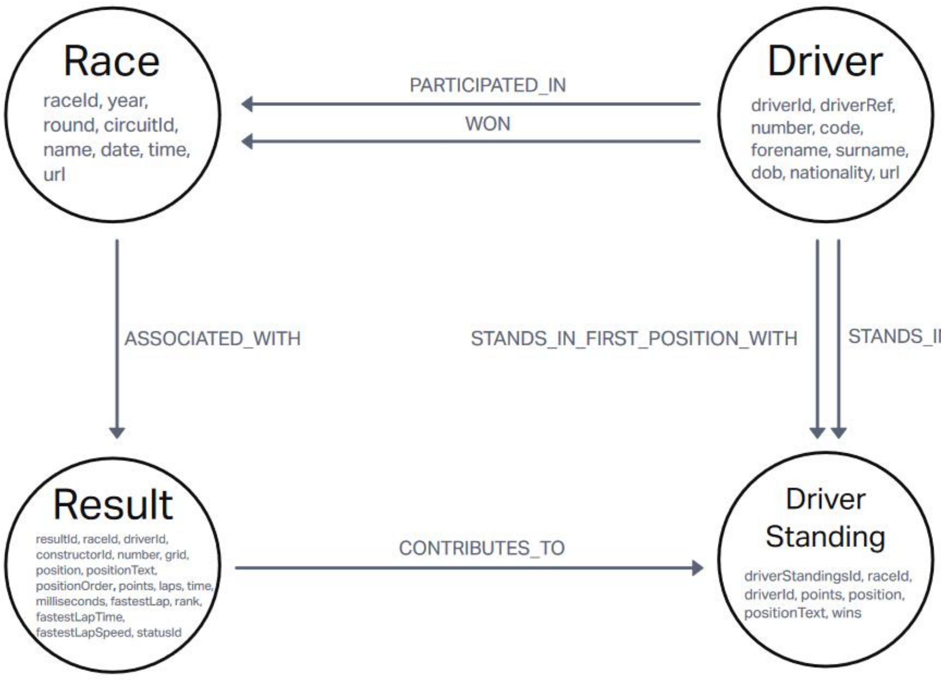
db.Driver.find({"nationality": "YOUR_SELECTION"}, {"forename": 1, "surname": 1, "WhichStandsIn.points": 1, "WhichStandsIn.position": 1})

db.Driver.find({"surname": " YOUR_SELECTION", "number": YOUR_SELECTION}, {"WhichStandsIn.driverStandingId": 1})

db.DriverStanding.find({"position": 1}, {"forename":1, "ContributedBy.resultId": 1})

Neo4J design

Logical schema


Race
raceId, year, round, circuitId, name, date, time, url
PARTICIPATED_IN
WON
Driver
driverId, driverRef, number, code, forename, surname, dob, nationality, url
ASSOCIATED_WITH
STANDS_IN_FIRST_POSITION_WITH
STANDS_IN
Result
resultId, raceId, driverId, constructorId, number, grid, position, positionText, positionOrder, points, laps, time, milliseconds, fastestLap, rank, fastestLapTime, fastestLapSpeed, statusId
CONTRIBUTES_TO
Driver Standing
driverStandingsId, raceId, driverId, points, position, positionText, wins

Queries

MATCH (driver:Driver {surname: 'Surname'})-[:PARTICIPATED_IN]->(race:Race)<-[:associated_with]-(result:Result)
RETURN race.raceId, result.resultId

MATCH(race:Race{name:'name' , date:'date'}) <- [WON] - (driver:Driver)
RETURN driver.surname, driver.nationality

MATCH(driver:Driver{nationality:"nationality"}) ->[STANDS_IN] -> (driverstanding:DriverStanding)
RETURN driver.forename, driver.surname, driverstandings.points, driverstandings.position

MATCH (driver:Driver{surname:"surname", number:"number"}) -> [STANDS_IN] -> (driverstandings:DriverStandings)
RETURN driverstandings.driverStandingId

MATCH(driver:Driver) ->[STANDS_IN_FIRST_POSITION_WITH]->(driverstanding:DriverStanding) <- [CONTRIBUTES_TO] -(result:Result)
RETURN: driver.forenamen, results

Schema design

With respect to the entity relationship model, we added two new relationships, [WON] and [STANDS_IN_FIRST_POSITION_WITH], to avoid accessing the properties of the node. This is important since the main property of the graph database is that the traversal is done in a constant time and adding a new relationship is encouraged to avoid the access to the nodes.

System identification discussion

We can choose between three different database paradigms: document based, columns based and graphs.

Document-based databases store data in documents, typically in JSON or BSON format. Each document is a self-contained unit with key-value pairs. The main advantage is the flexible schema that can handle semi structured data and nested structure that evolves, all of that comes at the cost of not so efficient relations traversal between documents.

Column-based databases store data in columns rather than rows. Each column is stored separately, allowing for efficient data retrieval and analysis. Firstly, they excel in analytical queries and aggregations, providing faster processing times and improved performance for complex analyses. Secondly, compression techniques can be applied to save storage space without compromising data integrity, which is especially beneficial for large datasets. Column-based databases are well-suited for read-heavy workloads, offering superior performance compared to traditional row-based databases. Moreover, they are scalable, capable of handling large volumes of data efficiently, and adapting to the growing needs of businesses. However, they may not be as efficient for write-heavy workloads or complex transactional processing, and schema changes can pose challenges. In summary, while column-based databases offer significant benefits in terms of performance, storage efficiency, and scalability, it is essential to consider specific application needs and potential limitations before adoption.

Graph databases offer a revolutionary approach to managing and querying complex data relationships through graph structures comprising nodes, edges, and properties. They excel in managing and querying data with intricate relationships, making them ideal for applications like social networks, fraud detection, and recommendation systems. Efficient traversal of relationships is a significant advantage, facilitating rapid access to connected data points. These databases offer schema flexibility, accommodating evolving relationships without constraints. However, they may not perform as well for simple, non-relational queries, and encounter storage and query optimization challenges with large graphs. Additionally, they may not be suitable for all data types, particularly when relationships are not the primary focus. In conclusion, while graph databases offer unparalleled capabilities for managing complex relationships, their suitability depends on specific use cases and data requirements.

As we said at the beginning, our choice is a document-based system. The main reason are: the CAP theorem and the possibility of having nested attributes. If we had chosen a column based or graph database their creation wouldn’t be possible.

Configuration details

We used MongoDB compass: firstly a database that contains one collection for each csv file has been created, then we used the aggregation functionality of the DB to create the aggregates that we identified before.

*For the creation of the "**DriverStanding**" aggregate we used:*

```
[
 {
  $lookup: {
    from: "Races",
    localField: "raceId",
    foreignField: "raceId",
    as: "race",
  },
 },
 {
  $unwind: "$race", // Since it is a one-to-one relationship, we can safely unwind.
 },
 {
  $lookup: {
    from: "Driver",
    localField: "driverId",
    foreignField: "driverId",
    as: "driver",
  },
 },
 {
  $unwind: "$driver", // Since it is a one-to-one relationship, we can safely unwind.
 },
 {
  $lookup: {
    from: "Result",
    let: {
      raceId: "$race.raceId",
      driverId: "$driverId",
    },
    pipeline: [
      {
       $match: {
         $expr: {
           $and: [
             {
              $eq: ["$raceId", "$$raceId"],
             },
             {
              $eq: [
                "$driverId",
                "$$driverId",
              ],
             },
           ],
         },
        },
      },
      {
        $project: {
          _id: 0,
          resultId: 1,
        },
      },
    ],
    as: "ContributedBy",
  },
},
{
  $project: {
    driverStandingsId: 1,
    forename: "$driver.forename",
    position: 1,
    ContributedBy: 1,
  },
}
]
```

*For the creation of the "**Result**" aggregate we used:*

```
[
 {
  $lookup: {
    from: "Driver",
    localField: "driverId",
    foreignField: "driverId",
    as: "driver",
  },
 },
 {
  $unwind: "$driver", // Since it is a one-to-one relationship, we can safely unwind.
 },
 {
  $lookup: {
    from: "Races",
    localField: "raceId",
    foreignField: "raceId",
    as: "race",
  },
 },
 {
  $unwind: "$race", // Since it is a one-to-one relationship, we can safely unwind.
 },
 {
  $project: {
    resultId: 1,
    position: 1,
    name: "$race.name",
    // Include name from the Race collection
```

```
      date: "$race.date",
      // Include date from the Race collection
      WithParticipants: {
        surname: "$driver.surname",
        nationality: "$driver.nationality",
      },
    },
  },
]
```

*For the creation of the "**Driver**" aggregate we used:*

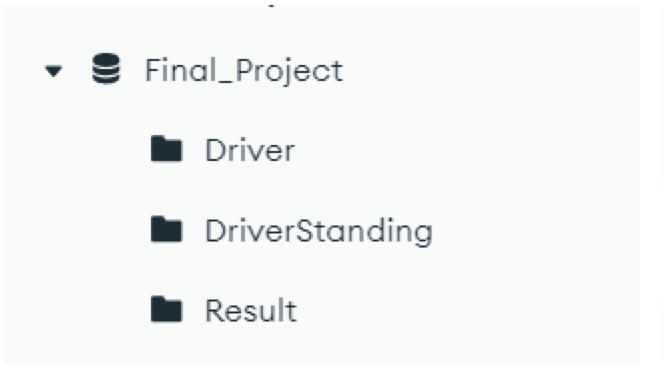
```
[

  {

    $lookup: {

      from: "DriverStanding",

      localField: "driverId",

      foreignField: "driverId",

      as: "driverStandings",

    },

  },

  {

    $lookup: {

      from: "Races",

      localField: "driverStandings.raceId",

      foreignField: "raceId",

      as: "races",

    },

  },

  {

    $lookup: {

      from: "Result",

      localField: "races.raceId",

      foreignField: "raceId",

      as: "results",

    },

  },
  {
    $project: {
      _id: 0,
      driverId: 1,
      number: 1,
      forename: 1,
      surname: 1,
      nationality: 1,
      WhichStandsIn: {
       $map: {
        input: "$driverStandings",
        as: "standing",
        in: {
         driverStandingsId:
          "$$standing.driverStandingsId",
         points: "$$standing.points",
         position: "$$standing.position",
        },
       },
      },
      ParticipatesIn: {
       $map: {
        input: "$races",
        as: "race",
        in: {
         raceId: "$$race.raceId",
         name: "$$race.name",
         date: "$$race.date",
         AssociatedWith: {
          $map: {
           input: {
            $filter: {
```

```
                    input: "$results",
                    as: "result",
                    cond: {
                      $eq: [
                        "$$result.raceId",
                        "$$race.raceId",
                      ], // Check for raceId equality
                    },
                  },
                },
                as: "filteredResult",
                in: {
                  resultId:
                    "$$filteredResult.resultId",
                },
              },
            },
          },
        },
      },
    },
  },
]
```

System configuration

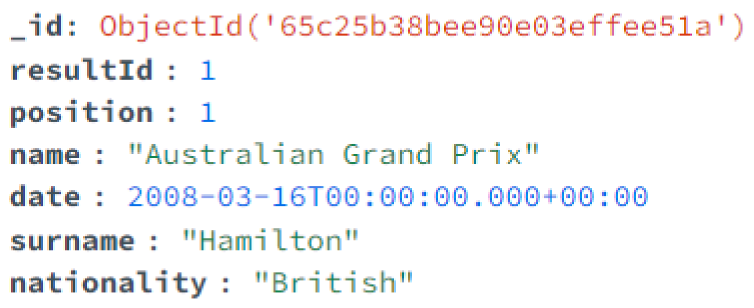
Our MongoDB compass database is composed of three collection: Driver, DriverStanding and Results, all created based on the methodology proposed during the course.

Logical schema



Result collection

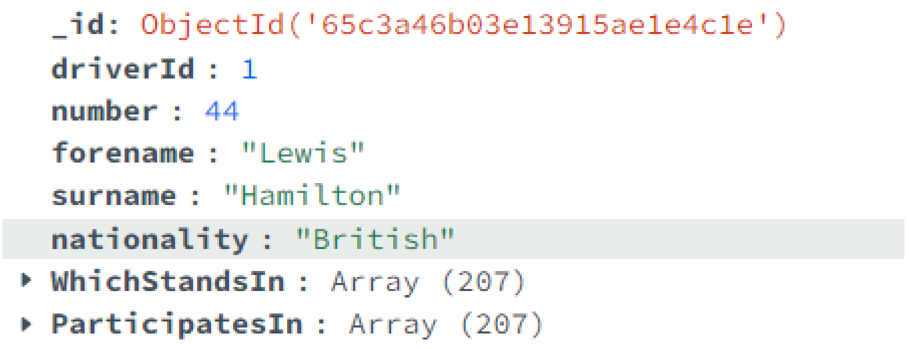
Composed by 23’700 documents like this:



```
_id: ObjectId('65c25b38bee90e03effee51a')
resultId : 1
position : 1
name : "Australian Grand Prix"
date : 2008-03-16T00:00:00.000+00:00
surname : "Hamilton"
nationality : "British"
```

Driver collection

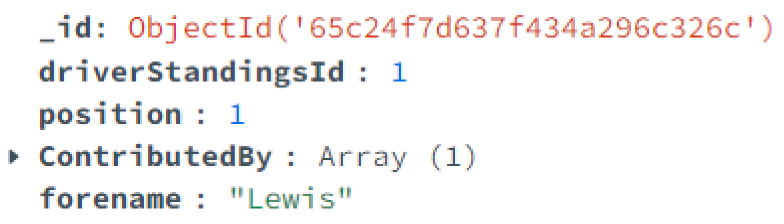
Composed by 842 documents like this:



```
  _id: ObjectId('65c3a46b03e13915ae1e4c1e')
  driverId : 1
  number : 44
  forename : "Lewis"
  surname : "Hamilton"
  nationality : "British"
▸ WhichStandsIn : Array (207)
▸ ParticipatesIn : Array (207)
```

DriverStanding collection

Composed by 31’000 documents like this:



```
  _id: ObjectId('65c24f7d637f434a296c326c')
  driverStandingsId : 1
  position : 1
▸ ContributedBy : Array (1)
  forename : "Lewis"
```

Workload on MongoDB

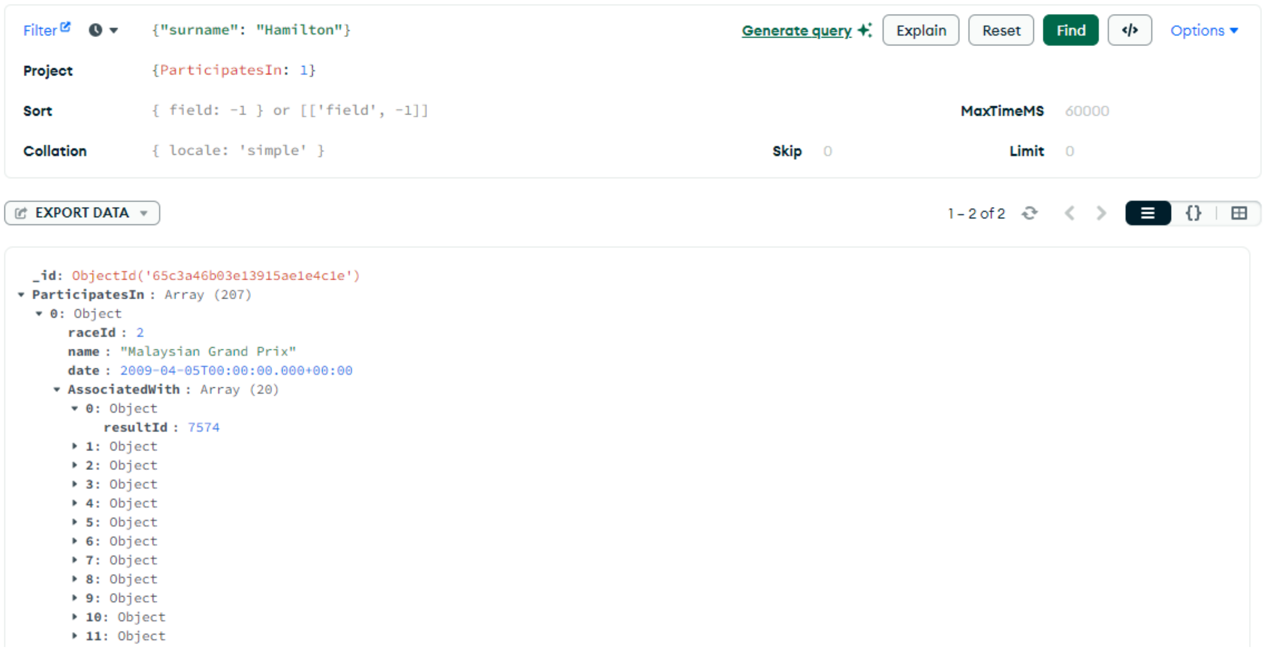
The workload is fully implemented in our MongoDB database and each query presented in the previous pages can be computed without any issue.

Here we provide the result of the first query:

Given a driver surname, return the races he participated in and their respective results.

We decided to search for drivers who have *Hamilton* surname. The query translates in:
db.Driver.find({"surname": "Hamilton"}, {ParticipatesIn: 1})

The database provides all the documents related to drivers called Hamilton along with a list containing all the races they took part and the respective results of that race:



```
Filter      {"surname": "Hamilton"}
Project     {ParticipatesIn: 1}
Sort        { field: -1 } or [['field', -1]]          MaxTimeMS  60000
Collation   { locale: 'simple' }          Skip  0     Limit      0

1 - 2 of 2

  _id: ObjectId('65c3a46b03e13915ae1e4c1e')
▾ ParticipatesIn : Array (207)
  ▾ 0: Object
      raceId : 2
      name : "Malaysian Grand Prix"
      date : 2009-04-05T00:00:00.000+00:00
    ▾ AssociatedWith : Array (20)
      ▾ 0: Object
          resultId : 7574
      ▸ 1: Object
      ▸ 2: Object
      ▸ 3: Object
      ▸ 4: Object
      ▸ 5: Object
      ▸ 6: Object
      ▸ 7: Object
      ▸ 8: Object
      ▸ 9: Object
      ▸ 10: Object
      ▸ 11: Object
```

Link to the video presentation:

1. https://drive.google.com/drive/folders/1doNL0CROFcUsBoKu5Tw9m7F4b5XZJ7Oa?usp=sharing