Database Project Report

Humbat Jamalov

Part 1

1. *Description of the Information System:*

The information system for the driving monitoring website is designed to track and analyze driving sessions under different real-world conditions. Each session is described by a combination of variables that describe the environment and context in which the session takes place. These variables include traffic conditions, road types, weather conditions, and visibility levels. The system stores these parameters for each session to evaluate the experience and quality of driving.

By gathering this information, the system enables meaningful insights into how external conditions affect driving behavior and performance. The entities representing each condition interact with the driving sessions, forming the foundation of data storage and analysis. At this stage, the exact nature of the relationships between these entities is not specified, but it is understood that each driving session is associated with one or more values from these condition-based entities.

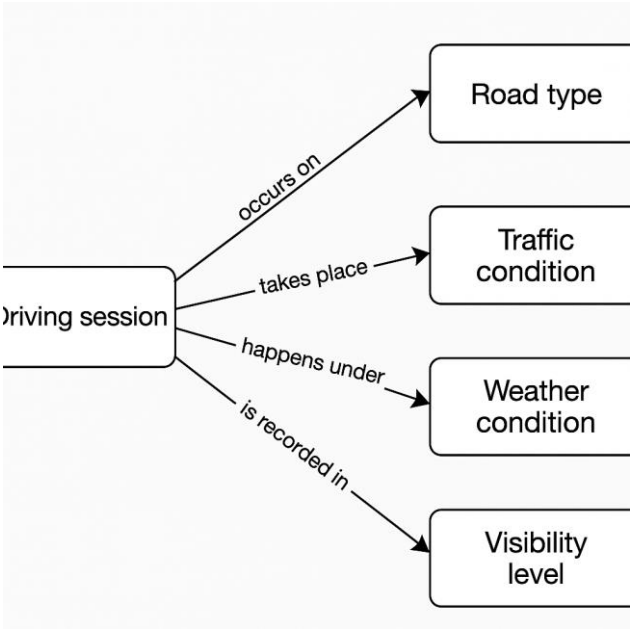
2. *Information system rules:*

- **Traffic Conditions:**
  Each driving session occurs under several traffic conditions such as light, moderate, or heavy traffic etc. The traffic condition can change during a session, therefore, each session is linked to multiple traffic conditions.
- **Road Type:**
  A driving session takes place on many types of roads such as urban, rural, gravel or highway etc. There can be many road types in one driving session.
- **Weather Condition:**
  Each driving session is associated with a specific weather condition, like sunny, rainy, or foggy etc. If

the weather changes, the system considers it a different session. Hence, each session is connected to one weather condition.

- **Visibility Level:**
  Visibility plays a crucial role in driving experience and safety. Each session is defined by a single visibility level such as clear, moderate, or low etc. This value is recorded as part of the session's environment.

3. CDM - Conceptual Data Model



4. Data Dictionary

Driving Session

Attribute Name	Description	Data Type	Size
session_id	Unique ID for driving session	Integer	11 digits, 4 bytes
start_date	Start time of session	Datetime	YYYY-MM-DD hh:mm:ss 8 bytes
end_date	End time of session	Datetime	YYYY-MM-DD hh:mm:ss 8

			bytes
mileage	Km covered	Float	11 digits, 4 bytes

Traffic Condition

Attribute Name	Description	Data Type	Size
traffic_condition_id	Unique ID for traffic condition	Integer	11 digits, 4 bytes
traffic_condition	Traffic condition during session	Varchar	20

Road Type

Attribute Name	Description	Data Type	Size
road_type_id	Unique ID for road type	Integer	11 digits, 4 bytes
road_type	Road type during session	Varchar	20

Weather Condition

Attribute Name	Description	Data Type	Size
weather_condition_id	Unique ID for weather condition	Integer	11 digits, 4 bytes
weather_condition	Weather condition during session	Varchar	20

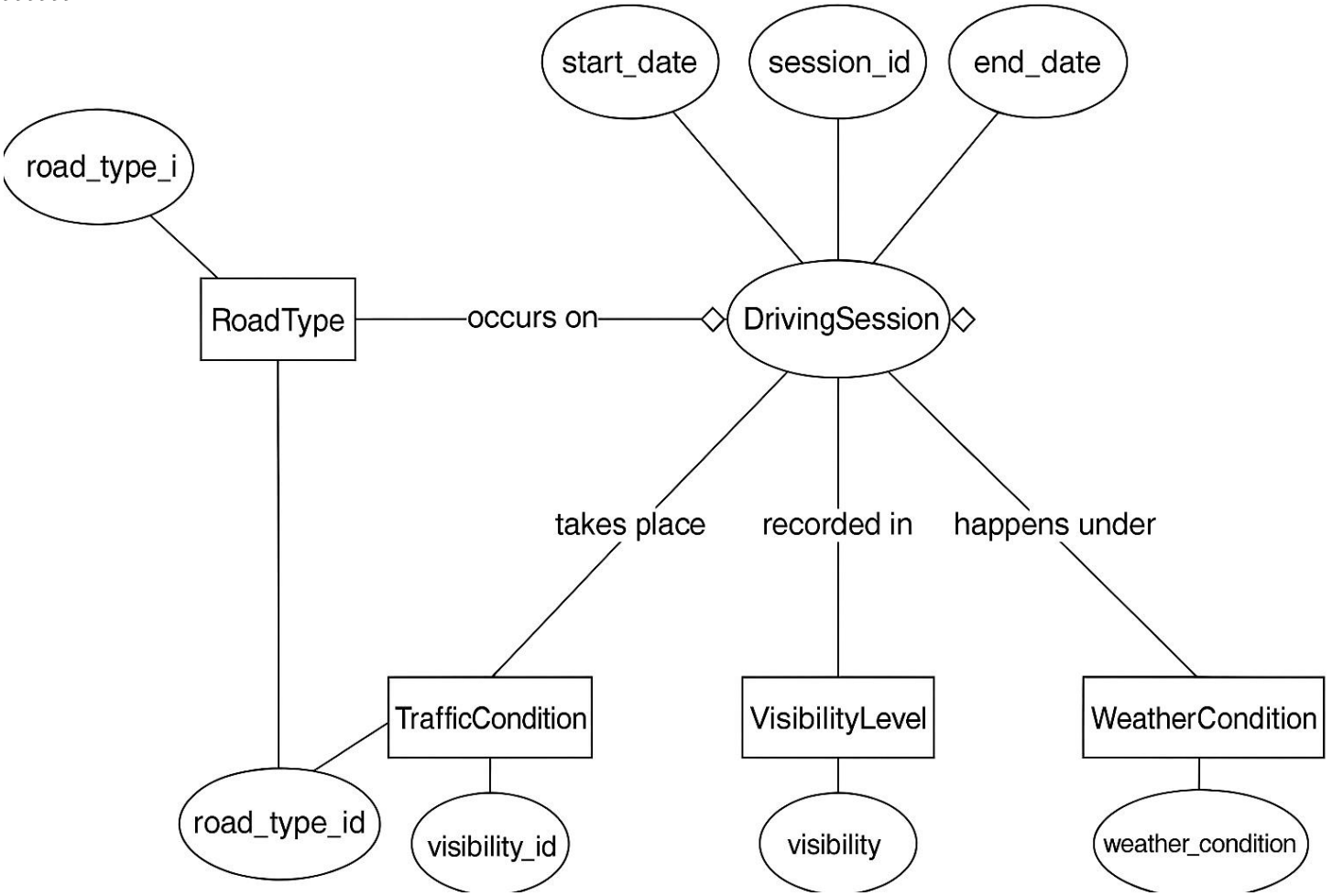
Visibility Level

Attribute Name	Description	Data Type	Size
visibility_id	Unique ID for visibility level	Integer	11 digits, 4 bytes
visibility	Visibility level during session	Varchar	20

The data dictionary outlines the key variables that describe driving sessions and their surrounding conditions. Each table represents a distinct component of the driving environment—such as weather, traffic, or road type—using consistent attribute names and standardized data types. Every entity includes a unique identifier to distinguish individual records. All attributes are functionally dependent on their corresponding identifiers, ensuring the data is normalized and maintains integrity.

5. ER Diagram (Logical Data Model)

yyyyyy

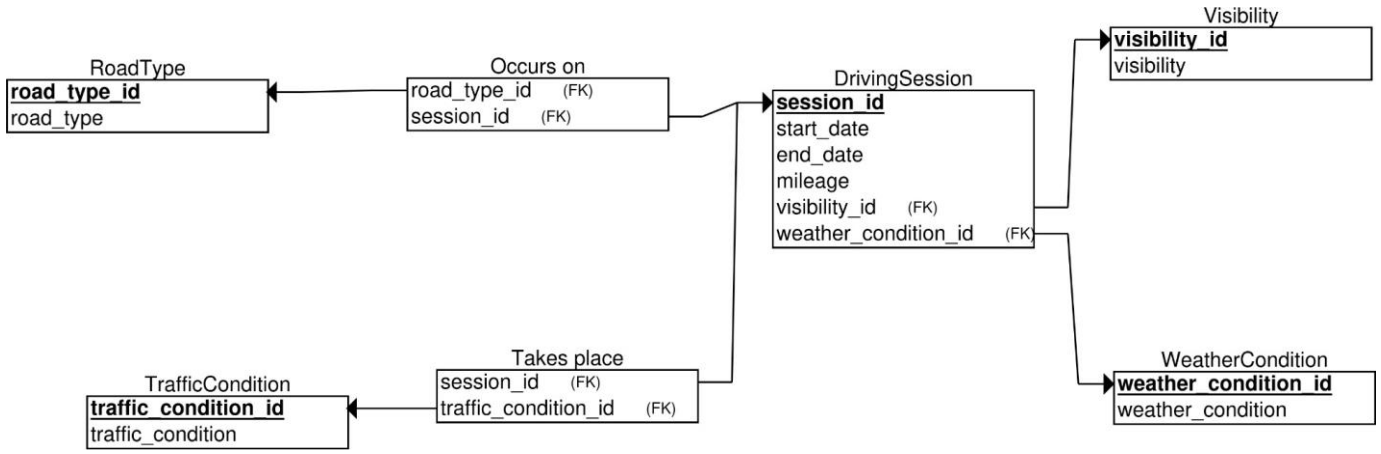


The Logical Data Model (LDM) depicts the core entities, their attributes, and the relationships among them. It clearly defines the types of relationships and their cardinalities within the model.

Cardinalities:

- One driving session may involve one or multiple traffic conditions.
- A single traffic condition may be linked to multiple driving sessions or none at all.
- A driving session may take place on one or several road types.
- A road type may be associated with multiple or no driving sessions.
- Each driving session is assigned exactly one visibility level.
- A visibility level can correspond to many or no driving sessions.
- Each driving session is linked to one specific weather condition.
- A weather condition may be present in many or no driving sessions.

6. PDM - Physical Data Model:



The Physical Data Model (PDM) is the last design of the project, the most detailed one that helps us create our database.

Primary keys:

- DrivingSession: session_id (uniquely identifies each driving experience)
- TrafficCondition: traffic_condition_id (acts as identifier for the traffic condition)
- RoadType: road_type_id (identifier for road type)
- Visibility: visibility_id (ensures uniqueness of each visibility level)
- WeatherCondition: weather_condition_id (unique identifier for weather)

Foreign Keys:

- Takes_place.session_id → DrivingSession.session_id (links the traffic condition to driving session, junction table for many-to-many relationship)

- Takes_place.traffic_condition_id → TrafficCondition.traffic_condition_id (other side of many-to-many relationship between DrivingSession and Traffic condition)
- Occurs_on.session_id → DrivingSession.session_id (links the road type condition to driving session, junction table for many-to-many relationship)
- Occurs_on.road_type_id → RoadType.road_type_id (other side of many-to-many relationship between DrivingSession and RoadType)
- DrivingSession.weather_condition_id → WeatherCondition.weather_condition_id (each experience will have its weather id)
- DrivingSession.visibility_id → Visibility.visibility_id (each experience will have its visibility id)

7. *SQL Representation of the Relation Scheme:*

Here is the SQL code based on the PDM diagram above:

```
CREATE TABLE RoadType
(
road_type_id INT NOT NULL,
road_type VARCHAR(50) NOT NULL,
PRIMARY KEY (road_type_id)
);

CREATE TABLE Visibility
(
visibility_id INT NOT NULL,
visibility VARCHAR(50) NOT NULL,
PRIMARY KEY (visibility_id)
);

CREATE TABLE WeatherCondition
(
weather_condition_id INT NOT NULL,
weather_condition VARCHAR(50) NOT NULL,
PRIMARY KEY (weather_condition_id)
);
CREATE TABLE TrafficCondition
(
traffic_condition_id INT NOT NULL,
traffic_condition VARCHAR(50) NOT NULL,
PRIMARY KEY (traffic_condition_id)
);
```

```
CREATE TABLE DrivingSession
(
session_id INT NOT NULL,
start_date TIMESTAMP NOT NULL,
end_date TIMESTAMP NOT NULL,
mileage FLOAT,
visibility_id INT NOT NULL,
weather_condition_id INT NOT NULL,
PRIMARY KEY (session_id),
FOREIGN KEY (visibility_id) REFERENCES Visibility(visibility_id),
FOREIGN KEY (weather_condition_id) REFERENCES WeatherCondition(weather_condition_id)
);

CREATE TABLE OccursOn
(
session_id INT NOT NULL,
road_type_id INT NOT NULL,
FOREIGN KEY (session_id) REFERENCES DrivingSession(session_id),
FOREIGN KEY (road_type_id) REFERENCES RoadType(road_type_id)
);

CREATE TABLE TakesPlace
(
session_id INT NOT NULL,
traffic_condition_id INT NOT NULL,
FOREIGN KEY (session_id) REFERENCES DrivingSession(session_id),
FOREIGN KEY (traffic_condition_id) REFERENCES TrafficCondition(traffic_condition_id)
);
```

Part 2 : Mongo Database

MongoDB: Step 1,2

First, we examine the JSON files that have been imported into our connected MongoDB cluster as collections. For instance, below is the content of the DrivingSession.json file:

```
[
  {
    "session_id": 1,
    "start_date": "2025-05-17T08:00:00",
    "end_date": "2025-05-17T08:30:00",
    "mileage": 15.2,
    "visibility_id": 1,
    "weather_condition_id": 1,
    "traffic_condition_ids": [1, 2],
    "road_type_ids": [1, 2]
  },
  {
    "session_id": 2,
    "start_date": "2025-05-18T14:10:00",
    "end_date": "2025-05-18T15:05:00",
    "mileage": 18.3,
    "visibility_id": 2,
    "weather_condition_id": 2,
    "traffic_condition_ids": [2],
    "road_type_ids": [2, 3]
  }
]
```


As shown in the screenshot, each driving session includes foreign keys referencing related attributes such as Visibility, Weather Condition, Road Type, and Traffic Condition. This design mirrors the structure of a relational (MySQL) database, which is typically normalized. Normalization promotes data consistency and reduces redundancy by storing related data in separate tables.

In contrast, a denormalized database structure embeds related data directly within the parent document, leading to nested documents and increased redundancy. Denormalized models do not require a strict schema, allowing flexibility—fields can be omitted without errors. In normalized systems like MySQL, this would result in errors due to schema mismatch. To overcome this, we use MongoDB, which is well-suited for managing denormalized structures.

In this phase, we aim to transform our initially normalized structure into a denormalized one using MongoDB tools such as mongosh.

Building the aggregation pipeline in MongoDB

This section explains how we transform our normalized database into a denormalized structure using an aggregation pipeline in MongoDB via the mongosh interface.

Step 1: Connecting to MongoDB via mongosh

We connect to our MongoDB cluster using mongosh. The connection step looks like this:

```
Current Mongosh Log ID: 682cce0f9c912d0949c59f34
Connecting to:          mongodb+srv://<credentials>@drivingexperiencecluste.0tf5uje.mon
godb.net/?appName=mongosh+2.5.1
Using MongoDB:          8.0.9 (API Version 1)
Using Mongosh:          2.5.1

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/


To help improve our products, anonymous usage data is collected and sent to MongoDB per
iodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

Atlas atlas-afr4dm-shard-0 [primary] test>
```

Step 2: Viewing Normalized Structure

Before performing aggregation, we review the normalized database structure so we can later compare it with the denormalized version. To do this, we use a set of commands to examine the imported collections, including DrivingSession, VisibilityLevel, WeatherCondition, RoadType, and TrafficCondition, which were loaded from their respective JSON files. Below is an example command used to inspect the DrivingSession collection:

```
Atlas atlas-afr4dm-shard-0 [primary] DrivingExperienceCluster> db.DrivingSession.find().pretty()
[
  {
    _id: ObjectId('682cd9eb9c912d0949c59f3d'),
    session_id: 1,
    start_date: '2025-05-17T08:00:00',
    end_date: '2025-05-17T08:30:00',
    mileage: 15.2,
    visibility_id: 1,
    weather_condition_id: 1,
    traffic_condition_ids: [ 1, 2 ],
    road_type_ids: [ 1, 2 ]
  },
  {
    _id: ObjectId('682cd9eb9c912d0949c59f3e'),
    session_id: 2,
    start_date: '2025-05-18T14:10:00',
    end_date: '2025-05-18T15:05:00',
    mileage: 18.3,
    visibility_id: 2,
    weather_condition_id: 2,
    traffic_condition_ids: [ 2 ],
    road_type_ids: [ 2, 3 ]
  }
]
Atlas atlas-afr4dm-shard-0 [primary] DrivingExperienceCluster>
```

All the commands:

db.DrivingSession.find().pretty()

db.TrafficCondition.find().pretty()

```
db.RoadType.find().pretty()
db.WeatherCondition.find().pretty()
db.VisibilityLevel.find().pretty()
```

Step 3: Applying the Aggregation Pipeline to normalized database

To create a single document combining all data from all related collections, we use the aggregation pipeline that uses multiple $lookup operations and transformations. Using the following code to execute the aggregation pipeline in mongosh:

```
db.DrivingSession.aggregate([
 {
        $lookup: {
        from: "RoadType",
        localField: "road_type_ids",
        foreignField: "road_type_id",
        as: "road_types"
        }
 },
 {
        $lookup: {
        from: "TrafficCondition",
        localField: "traffic_condition_ids",
        foreignField: "traffic_condition_id",
        as: "traffic_conditions"
        }
 },
 {
        $lookup: {
        from: "WeatherCondition",
        localField: "weather_condition_id",
        foreignField: "weather_condition_id",
        as: "weather"
        }
 },
 {
        $unwind: "$weather"
 },
 {
        $lookup: {
        from: "VisibilityLevel",
```

```
        localField: "visibility_id",
        foreignField: "visibility_id",
        as: "visibility"
        }
  },
  {
        $unwind: "$visibility"
  },
  {
        $project: {
        _id: 0,
        session_id: 1,
        start_date: 1,
        end_date: 1,
        mileage: 1,
        road_types: 1,
        traffic_conditions: 1,
        weather: 1,
        visibility: 1
        }
  }
])
```

How does the code work?

I used the Aggregation Pipeline to denormalize the DrivingSession collection by joining it with four related collections: RoadType, WeatherCondition, VisibilityLevel, and TrafficCondition. After combining them into a single collection, the related data became nested, with all details from the child collections encapsulated within each parent DrivingSession document.

`$lookup` → performs an operation equivalent to `LEFT JOIN` as in relational databases, but for collections.

`$unwind` → turns an array field in a JSON file into individual documents — one document per array element.

`$project` → selects and reshapes fields in the output. Similar to the `SELECT` clause in SQL (controls what to include or exclude).

Step 4: The Final Denormalized Structure

Using the aggregation pipeline, we ended up encapsulating embedded fields from all related collections that are describing the DrivingSession:

```
[
  {
    session_id: 1,
    start_date: '2025-05-17T08:00:00',
    end_date: '2025-05-17T08:30:00',
    mileage: 15.2,
    road_types: [
      {
        _id: ObjectId('682cd9f69c912d0949c59f43'),
        road_type_id: 1,
        road_type: 'Urban'
      },
      {
        _id: ObjectId('682cd9f69c912d0949c59f44'),
        road_type_id: 2,
        road_type: 'Highway'
      }
    ],
    traffic_conditions: [
      {
        _id: ObjectId('682cd9f09c912d0949c59f3f'),
        traffic_condition_id: 1,
        traffic_condition: 'Light'
      },
      {
        _id: ObjectId('682cd9f09c912d0949c59f40'),
        traffic_condition_id: 2,
        traffic_condition: 'Moderate'
      }
    ],
    weather: {
      _id: ObjectId('682cd9fb9c912d0949c59f47'),
      weather_condition_id: 1,
      weather_condition: 'Sunny'
    },
    visibility: {
      _id: ObjectId('682cda029c912d0949c59f4b'),
      visibility_id: 1,
      visibility: 'Clear'
    }
  },
```

```
  {
    session_id: 2,
    start_date: '2025-05-18T14:10:00',
    end_date: '2025-05-18T15:05:00',
    mileage: 18.3,
    road_types: [
      {
        _id: ObjectId('682cd9f69c912d0949c59f44'),
        road_type_id: 2,
        road_type: 'Highway'
      },
      {
        _id: ObjectId('682cd9f69c912d0949c59f45'),
        road_type_id: 3,
        road_type: 'Rural'
      }
    ],
    traffic_conditions: [
      {
        _id: ObjectId('682cd9f09c912d0949c59f40'),
        traffic_condition_id: 2,
        traffic_condition: 'Moderate'
      }
    ],
    weather: {
      _id: ObjectId('682cd9fb9c912d0949c59f48'),
      weather_condition_id: 2,
      weather_condition: 'Rainy'
    },
    visibility: {
      _id: ObjectId('682cda029c912d0949c59f4c'),
      visibility_id: 2,
      visibility: 'Moderate'
    }
  }
]
```

Conclusion

In this project, I successfully demonstrated the complete cycle of relational database modeling using both MySQL and MongoDB. Beginning with conceptual modeling, I developed normalized relational schemas and implemented them as a functional SQL database with relevant queries. I then imported the structured data into MongoDB and used an aggregation pipeline to convert it into a denormalized format suitable for document-based querying. This process highlighted the strengths of normalization in ensuring data consistency, as well as the flexibility and performance benefits of denormalization in NoSQL environments. Overall, the project deepened my understanding of data modeling concepts across different database systems.